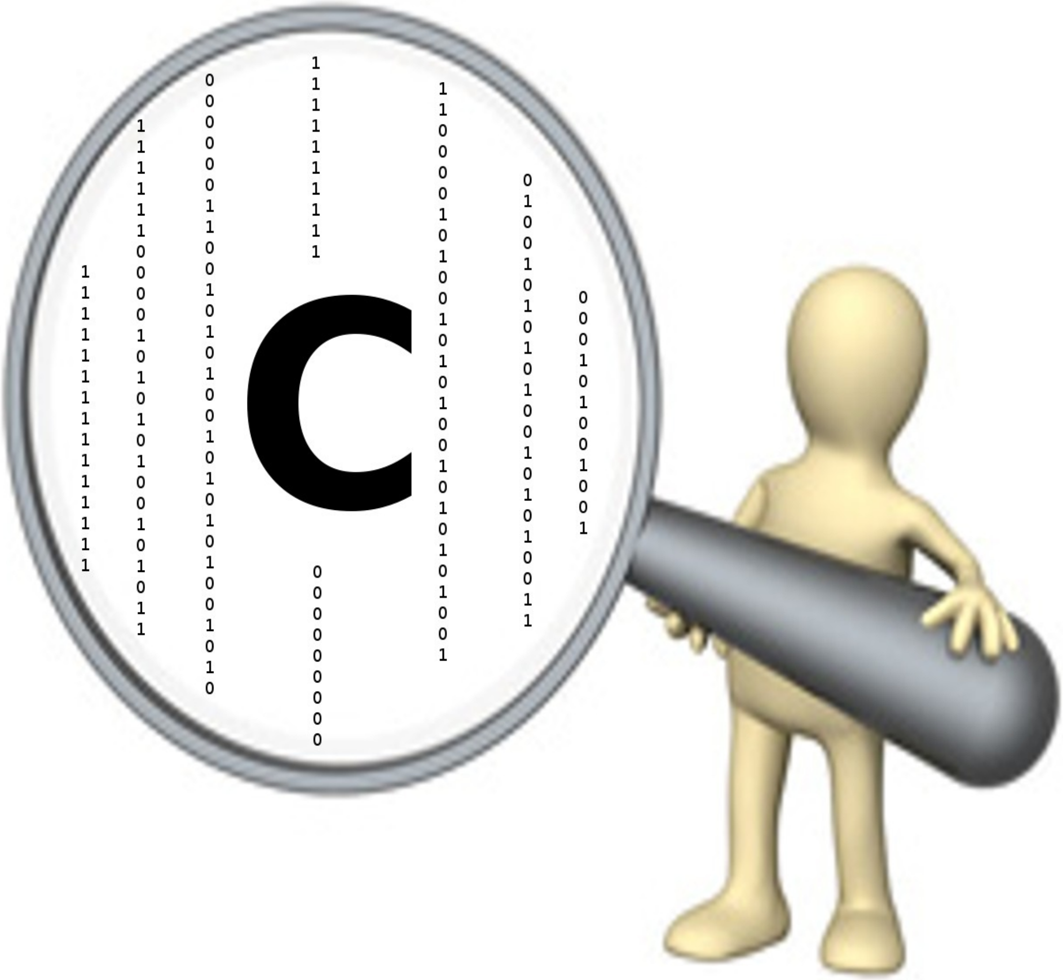
C 语言编程透视



@泰晓科技 TinyLab.org

目錄

> 联系作者：笔者为一位重度开源践行者，可微信联系：lzufalcon
> 实验云台：在线学 Linux，Linux 0.11，汇编，Shell，C ...
> 本书来源：开源书籍：C 语言编程透视 (by 泰晓科技)
> 报名参与：*Star/fork* GitHub 仓库 并发送 *Pull Request*
> 关注我们：扫描二维码 关注 @泰晓科技 微博和微信公众号
> 赞助我们：捐助我们, 更多原创开源书籍需要您的支持 ^o^

C 语言编程透视

v 0.2

本书与《深入浅出 Hello World》有着类似的心路历程，旨在以实验的方式去探究类似 `Hello World` 这样的小程序在开发与执行过程中的微妙变化，一层层揭开 C 语言程序开发过程的神秘面纱，透视背后的秘密，不断享受醍醐灌顶的美妙。

介绍

- 项目首页：http://www.tinylab.org/open-c-book
- 代码仓库：https://github.com/tinyclub/open-c-book
- 在线阅读：http://tinylab.gitbooks.io/cbook
- 实验云台：在线学 Linux，Linux 0.11，汇编，Shell，C ...

  更多背景和计划请参考：前言。

编译

要编译本书，请使用 Markdown Lab。

纠错

欢迎大家指出不足，如有任何疑问，请邮件联系 wuzhangjin at gmail dot com 或者直接修复并提交 Pull Request。

版权

关注我们

- 新浪微博

- 微信公众号

赞助我们

- 微信扫码赞助原创

- 访问 泰晓开源小店 支持心仪项目

更多原创开源书籍

- Shell 编程范例
- 嵌入式 Linux 知识库(eLinux.org 中文版)
- Linux 内核文档(Linux Documentation/ 中文版)

版本修订历史

Revision	Author	From	Date	Description
0.2	@吴章金 falcon	@泰晓科技	2015/07/23	调整格式，修复链接
0.1	@吴章金 falcon	@泰晓科技	2014/01/19	初稿

前言

- 背景
- 现状
- 计划

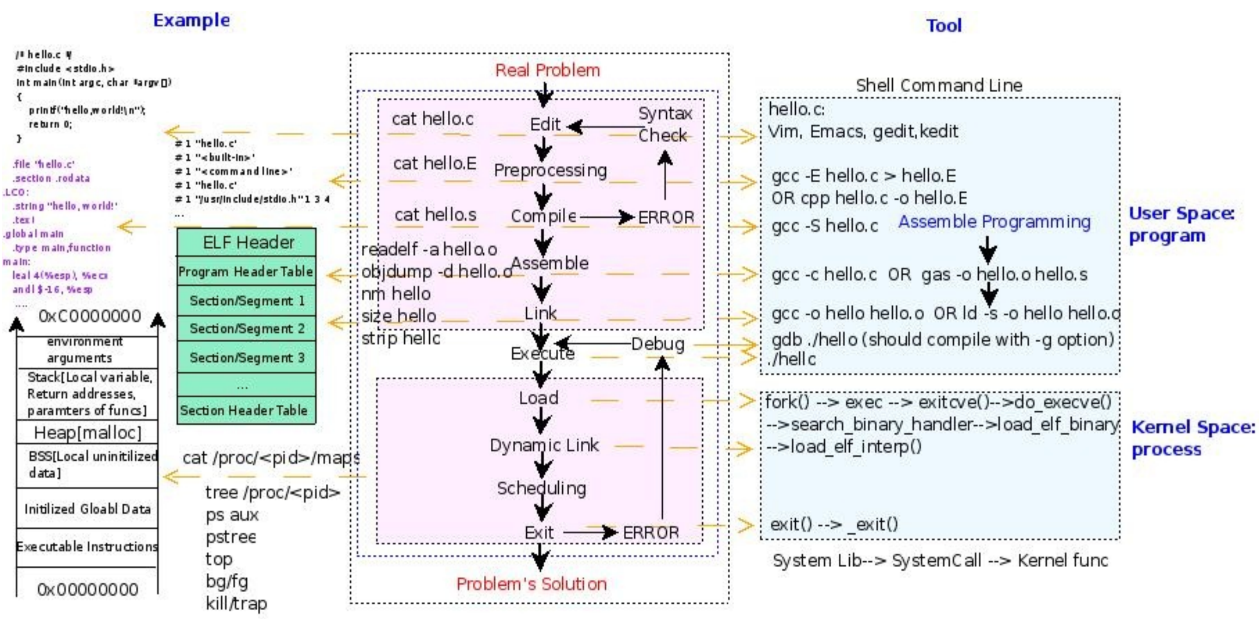
背景

笔者 2007 年开始系统地学习 Shell 编程，并在兰大开源社区写了序列文章。

在编写《Shell 编程范例》文章的《进程操作》一章时，为了全面了解进程的来龙去脉，对程序开发过程的细节、ELF 格式的分析、进程的内存映像等进行了全面地梳理，后来搞得“雪球越滚越大”，甚至脱离了 Shell 编程关注的内容。所以想了个小办法，“大事化小，小事化了”，把涉及到的内容进行了分解，进而演化成另外一个完整的序列。

2008 年 3 月 1 日，当初步完成整个序列时，做了如下的小结：

> 到今天，关于"Linux 下 C 语言开发过程"的一个简单视图总算粗略地完成了，从寒假之前的一段时间到现在过了将近一个月左右吧。写这个主题的目的源自“Shell 编程范例之进程操作”，当写到这一章时，突然对进程的由来、本身和去向感到“迷惑不解”。所以想着好好花些时间来弄清楚它们，现在发现，这个由来就是这里的程序开发过程，进程来自一个普通的文本文件，在这里是 C 语言程序，C 语言程序经过编辑、预处理、编译、汇编、链接、执行而成为一个进程；而进程本身呢？当一个可执行文件被执行以后，有了 exec 调用，被程序解释器映射到了内存中，有了它的内存映像；而进程的去向呢？通过不断地执行指令和内存映像的变化，进程完成着各项任务，等任务完成以后就可以退出了（exit）。
>
> 这样一份视图实际上是在寒假之前绘好的，可以从下图中看到它；不过到现在才明白背后的很多细节。这些细节就是这个序列的每个篇章，可以对照“视图”来阅读它们。



C Language Programming Procedure in Linux

现状

目前整个序列大部分都已经以 Blog 的形式写完，大体结构目下：

- 《把 VIM 打造成源代码编辑器》
  - 源代码编辑过程：用 VIM 编辑代码的一些技巧
  - 更新时间：2008-2-22
- 《GCC 编译的背后》
  - 编译过程：预处理、编译、汇编、链接
  - 第一部分：《预处理和编译》（更新时间：2008-2-22）
  - 第二部分：《汇编和链接》（更新时间：2008-2-22）
- 《程序执行的那一刹那 》
  - 执行过程：当从命令行输入一个命令之后
  - 更新时间：2008-2-15
- 《进程的内存映像》
  - 进程加载过程：程序在内存里是个什么样子？
  - 第一部分（讨论“缓冲区溢出和注入”问题）（更新时间：2008-2-13）
  - 第二部分（讨论进程的内存分布情况）（更新时间：2008-6-1）
- 《进程和进程的基本操作》
  - 进程操作：描述进程相关概念和基本操作
  - 更新时间：2008-2-21

- 《动态符号链接的细节》
  - 动态链接过程：函数 puts/printf 的地址在哪里？
  - 更新时间：2008-2-26
- 《打造史上最小可执行ELF文件》
  - ELF 详解：从"减肥"的角度一层一层剖开 ELF 文件，最终获得一个可打印 Hello World 的 **45** 字节 ELF 可执行文件
  - 更新时间：2008-2-23
- 《代码测试、调试与优化小结》
  - 程序开发过后：内存溢出了吗？有缓冲区溢出？代码覆盖率如何测试呢？怎么调试汇编代码？有哪些代码优化技巧和方法呢？
  - 更新时间：2008-2-29

计划

考虑到整个 Linux 世界的蓬勃发展，Linux 和 C 语言的应用环境越来越多，相关使用群体会不断增加，所以最近计划把该序列重新整理，以自由书籍的方式不断更新，以便惠及更多的读者。

打算重新规划、增补整个序列，并以开源项目的方式持续维护，并通过 泰晓科技|TinyLab.org 平台接受读者的反馈，直到正式发行出版。

自由书籍将会维护在 泰晓科技 的项目仓库中。项目相关信息如下：

- 项目首页：http://www.tinylab.org/open-c-book/
- 代码仓库：https://github.com/tinyclub/open-c-book.git

欢迎大家指出本书初稿中的不足，甚至参与到相关章节的写作、校订和完善中来。

如果有时间和兴趣，欢迎参与。可以通过 泰晓科技 联系我们，或者直接关注微博 @泰晓科技并私信我们。

把 Vim 打造成源代码编辑器

前言

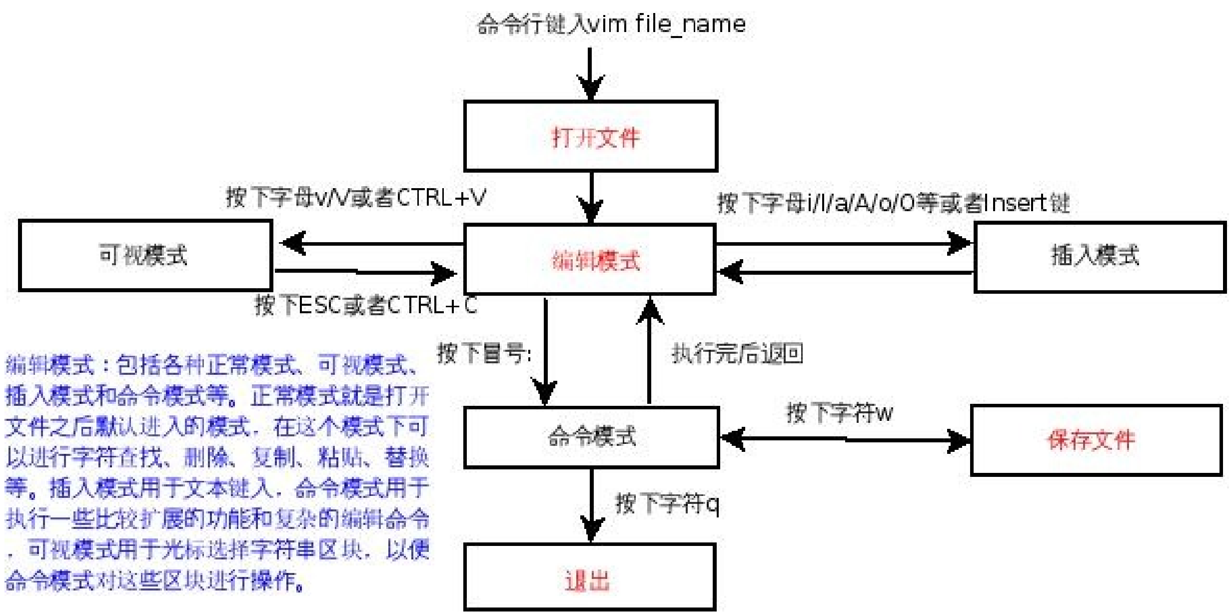
程序开发过程中，源代码的编辑主要是为了实现算法，结果则是一些可阅读的、便于检错的、可移植的文本文件。如何产生一份良好的源代码，这不仅需要一些良好的编辑工具，还需要开发人员养成良好的编程修养。

Linux 下有很多优秀的程序编辑工具，包括专业的文本编辑器和一些集成开发环境（IDE）提供的编辑工具，前者的代表作有 Vim 和 Emacs，后者的代表作则有 Eclipse，Kdevelop，Anjuta 等，这里主要介绍 Vim 的基本使用和配置。

常规操作

通过 Vim 进行文本编辑的一般过程包括：文件的打开、编辑、保存、关闭/退出，而编辑则包括插入新内容、替换已有内容、查找内容，还包括复制、粘贴、删除等基本操作。

该过程如下图：



下面介绍几个主要操作：

打开文件

在命令行下输入 `vim 文件名` 即可打开一个新文件并进入 Vim 的“编辑模式”。

编辑模式可以切换到命令模式（按下字符 `:` ）和插入模式（按下字母 `a/A/i/I/o/O/s/S/c/C` 等或者 Insert 键）。

编辑模式下，Vim 会把键盘输入解释成 Vim 的编辑命令，以便实现诸如字符串查找(按下字母 `/`)、文本复制（按下字母 `yy` ）、粘贴（按下字母 `pp` ）、删除（按下字母 `d` 等）、替换（ `s` ）等各种操作。

当按下 `a/A/i/I/o/O/s/S/c/C` 等字符时，Vim 先执行这些字符对应命令的动作（比如移动光标到某个位置，删除某些字符），然后进入插入模式；进入插入模式后可以通过按下 ESC 键或者是 `CTRL+C` 返回到编辑模式。

在编辑模式下输入冒号 `:` 后可进入命令模式，通过它可以完成一些复杂的编辑功能，比如进行正则表达式匹配替换，执行 Shell 命令（按下 `!` 命令）等。

实际上，无论是插入模式还是命令模式都是编辑模式的一种。而编辑模式却并不止它们两个，还有字符串查找、删除、替换等。

需要提到的是，如果在编辑模式按下字母 `v/V` 或者是 `CTRL+V` ，可以用光标选择一个区块，进而结合命令模式对这一个区块进行特定的操作。

编辑文件

打开文件以后即可进入编辑模式，这时可以进行各种编辑操作，包括插入、复制、删除、替换字符。其中两种比较重要的模式经常被“独立”出来，即上面提到的插入模式和命令模式。

保存文件

在退出之前需切换到命令模式，输入命令 `w` 以便保存各种编辑后的内容，如果想取消某种操作，可以用 `u` 命令。如果打开 Vim 编辑器时没有设定文件名，那么在按下 `w` 命令时会提示没有文件名，此时需要在 `w` 命令后加上需要保存的文件名。

退出/关闭

保存好内容后就可退出，只需在命令模式下键入字符 `q` 。如果对文件内容进行了编辑，却没有保存，那么 Vim 会提示，如果不想保存之前的编辑动作，那么可按下字符 `q` 并且在之后跟上一个感叹号 `!` ，这样会强制退出，不保存最近的内容变更。

命令模式

这里需要着重提到的是 Vim 的命令模式，它是 Vim 扩展各种新功能的接口，用户可以通过它启用和撤销某个功能，开发人员则可通过它为用户提供新的功能。下面主要介绍通过命令模式这个接口定制 Vim 以便我们更好地进行源代码的编辑。

编码风格与 indent 命令

先提一下编码风格。刚学习编程时，代码写得很“难看”（不方便阅读，不方便检错，看不出任何逻辑结构），常常导致心情不好，而且排错也很困难，所以逐渐意识到代码编写需要规范，即养成良好的编码风格，如果换成俗话，那就是代码的排版，让代码好看一些。虽说“编程的“（高雅一些则称开发人员）不一定懂艺术，不过这个应该不是“搞艺术的”（高雅一些应该是文艺工作人员）的特权，而是我们应该具备的专业素养。在 Linux 下，比较流行的“行业”风格有 KR 的编码风格、GNU 的编码风格、Linux 内核的编码风格（基于 KR 的，缩进是 8 个空格）等，它们都可以通过 `indent` 命令格式化，对应的选项分别是 `-kr` ， `-gnu` ， `-kr -i8` 。下面演示用 `indent` 命令把代码格式化成上面的三种风格。

这样糟糕的编码风格看着会让人想“哭”，太难阅读啦：

```
$ cat > test.c
/* test.c -- a test program for using indent */
#include<stdio.h>

int main(int argc, char *argv[])
{
 int i=0;
 if (i != 0) {i++; }
 else {i--; };
 for(i=0;i<5;i++)j++;
 printf("i=%d,j=%d\n",i,j);

 return 0;
}
```

格式化成 KR 风格，好看多了：

```
$ indent -kr test.c
$ cat test.c
/* test.c -- a test program for using indent */
#include<stdio.h>

int main(int argc, char *argv[])
{
    int i = 0;
    if (i != 0) {
        i++;
    } else {
        i--;
    };
    for (i = 0; i < 5; i++)
        j++;
    printf("i=%d,j=%d\n", i, j);
    return 0;
}
```

采用 GNU 风格，感觉不如 KR 的风格，处理 `if` 语句时增加了代码行，却并没明显改进效果：

```
$ indent -gnu test.c
$ cat test.c
/* test.c -- a test program for using indent */
#include<stdio.h>

int
main (int argc, char *argv[])
{
  int i = 0;
  if (i != 0)
    {
      i++;
    }
  else
    {
      i--;
    };
  for (i = 0; i < 5; i++)
    j++;
  printf ("i=%d,j=%d\n", i, j);
  return 0;
}
```

实际上 `indent` 命令有时候会不靠谱，也不建议“先污染再治理”，而是从一开始就坚持“可持续发展”的观念，在写代码时就逐步养成良好的风格。

需要提到地是，Linux 的编码风格描述文件为内核源码下的 Documentation/CodingStyle，而相应命令为 scripts/Lindent。

用 Vim 命令养成良好编码风格

从演示中可看出编码风格真地很重要，但是如何养成良好的编码风格呢？经常练习，遵守某个编码风格，一如既往。不过这还不够，如果没有一个好编辑器，习惯也很难养成。而 Vim 提供了很多辅助我们养成良好编码习惯的功能，这些都通过它的命令模式提供。现在分开介绍几个功能；

Vim 命令	功效
`:syntax on`	语法加“靓”（亮）
`:syntax off`	语法不加“靓”（亮）
`:set cindent`	C 语言自动缩进（可简写为 `set cin` ）
`:set sw=8`	自动缩进宽度（需要 `set cin` 才有用）
`:set ts=8`	设定 TAB 宽度
`:set number`	显示行号
`:set nonumber`	不显示行号
`:setsm`	括号自动匹配

这几个命令对代码编写来说非常有用，可以考虑把它们全部写到 `~/.vimrc` 文件（Vim 启动时会去加载这个文件里头的内容）中，如：

```
$ cat ~/.vimrc
:set number
:set sw=8
:set ts=8
:set sm
:set cin
:syntax on
```

相关小技巧

需要补充的几个技巧有；

- 对注释自动断行
    - 在编辑模式下，可通过 `gqap` 命令对注释自动断行（每行字符个数可通过命令模式下的 `set textwidth=个数` 设定）
- 跳到指定行
    - 命令模式下输入数字可以直接跳到指定行，也可在打开文件时用 `vim +数字 文件名` 实现相同的功能。
- 把 C 语言输出为 html
    - 命令模式下的 `TOhtml` 命令可把 C 语言输出为 html 文件，结合 `syntax`

`on` ，可产生比较好的网页把代码发布出去。

- 注释掉代码块
    - 先切换到可视模式（编辑模式下按字母 `v` 可切换过来），用光标选中一片代码，然后通过命令模式下的命令 `s#^#//#g` 把某这片代码注释掉，这非常方便调试某一片代码的功能。
- 切换到粘贴模式解决 Insert 模式自动缩进的问题
    - 命令模式下的 `set paste` 可解决复制本来已有缩进的代码的自动缩进问题，后可执行 `set nopaste` 恢复自动缩进。
- 使用 Vim 最新特性
    - 为了使用最新的 Vim 特性，可用 `set nocp` 取消与老版本的 Vi 的兼容。
- 全局替换某个变量名
    - 如发现变量命名不好，想在整个代码中修改，可在命令模式下用 `%s#old_variable#new_variable#g` 全局替换。替换的时注意变量名是其他变量一部分的情况。如果希望将变量"abc"全部替换成"xyz"又不希望把"abcd"错误替换成"xyzd",则可以在查找时指定边界: `%s#\<old_variable\>#new_variable#g` 。
- 把缩进和 TAB 键都替换为空格
    - 可考虑设置 `expandtab` ，即 `set et` ，如果要把以前编写的代码中的缩进和 TAB 键都替换掉，可以用 `retab` 。
- 关键字自动补全
    - 输入一部分字符后，按下 `CTRL+P` 或者 `CTRL+N` 即可。比如先输入 `prin` ，然后按下 `CTRL+P/N` 就可以补全了。
- 在编辑模式下查看手册
    - 可把光标定位在某个函数，按下 `Shift+k` 就可以调出 `man` ，很有用。
- 删除空行
    - 在命令模式下输入 `g/^$/d` ，前面 `g` 命令是扩展到全局，中间是匹配空行，后面 `d` 命令是执行删除动作。用替换也可以实现，键入 `%s#^\n##g` ，意思是把所有以换行开头的行全部替换为空。类似地，如果要把多个空行转换为一个可以输入 `g/^\n$/d` 或者 `%s#^\n$##g` 。
- 创建与使用代码交叉引用

    - 注意利用一些有用的插件，比如 `ctags` , `cscope` 等，可以提高代码阅读、分析的效率。特别是开放的软件。

- 回到原位置
    - 在用 `ctags` 或 `cscope` 时，当找到某个标记后，又想回到原位置，可按下 `CTRL+T` 。

这里特别提到 `cscope` ，为了加速代码的阅读，还可以类似上面在 `~/.vimrc` 文件中通过 `map` 命令预定义一些快捷方式，例如：

```
if has("cscope")
          set csprg=/usr/bin/cscope
          set csto=0
          set cst
          set nocsverb
          " add any database in current directory
          if filereadable("cscope.out")
            cs add cscope.out
          " else add database pointed to by environment
          elseif $CSCOPE_DB != ""
            cs add $CSCOPE_DB
          endif
          set csverb
:map \ :cs find g <C-R>=expand("<cword>")<CR><CR>
:map s :cs find s <C-R>=expand("<cword>")<CR><CR>
:map t :cs find t <C-R>=expand("<cword>")<CR><CR>
:map c :cs find c <C-R>=expand("<cword>")<CR><CR>
:map C :cs find d <C-R>=expand("<cword>")<CR><CR>
:map f :cs find f <C-R>=expand("<cword>")<CR><CR>
endif
```

因为 `s,t,c,C,f` 这几个 Vim 的默认快捷键用得不太多，所以就把它们给作为快捷方式映射了，如果已经习惯它们作为其他的快捷方式就换别的字符吧。

注 上面很多技巧中用到了正则表达式，关于这部分请参考：正则表达式 30 分钟入门教程。

更多的技巧可以看看后续资料。

后记

实际上，在源代码编写时还有很多需要培养的“素质”，例如源文件的开头注释、函数的注释，变量的命名等。这方面建议看看参考资料里的编程修养、内核编码风格、网络上流传的《华为编程规范》，以及《C Traps & Pitfalls》，《C-FAQ》等。

参考资料

- Vim 官方教程，在命令行下键入 vimtutor 即可
- vim 实用技术序列
  - 实用技巧
  - 常用插件
  - 定制 Vim
- Graphical vi-vim Cheat Sheet and Tutorial
- Documentation/CodingStyle
- scripts/Lindent。
- 正则表达式 30 分钟入门教程
- 也谈 C 语言编程风格：完成从程序员到工程师的蜕变
- Vim 高级命令集锦
- 编程修养
- C Traps & Pitfalls
- C FAQ

Gcc 编译的背后

前言

平时在 Linux 下写代码，直接用 `gcc -o out in.c` 就把代码编译好了，但是这背后到底做了什么呢？

如果学习过《编译原理》则不难理解，一般高级语言程序编译的过程莫过于：预处理、编译、汇编、链接。

`gcc` 在后台实际上也经历了这几个过程，可以通过 `-v` 参数查看它的编译细节，如果想看某个具体的编译过程，则可以分别使用 `-E`，`-S`，`-c` 和 `-O`，对应的后台工具则分别为 `cpp`，`cc1`，`as`，`ld`。

下面将逐步分析这几个过程以及相关的内容，诸如语法检查、代码调试、汇编语言等。

预处理

简述

预处理是 C 语言程序从源代码变成可执行程序的第一步，主要是 C 语言编译器对各种预处理命令进行处理，包括头文件的包含、宏定义的扩展、条件编译的选择等。

以前没怎么“深入”预处理，脑子对这些东西总是很模糊，只记得在编译的基本过程（词法分析、语法分析）之前还需要对源代码中的宏定义、文件包含、条件编译等命令进行处理。这三类的指令很常见，主要有 `#define`，`#include` 和 `#ifdef ... #endif`，要特别地注意它们的用法。

`#define` 除了可以独立使用以便灵活设置一些参数外，还常常和 `#ifdef ... #endif` 结合使用，以便灵活地控制代码块的编译与否，也可以用来避免同一个头文件的多次包含。关于 `#include` 貌似比较简单，通过 `man` 找到某个函数的头文件，复制进去，加上 `<>` 就好。这里虽然只关心一些技巧，不过预处理还是隐藏着很多潜在的陷阱（可参考《C Traps & Pitfalls》）也是需要注意的。下面仅介绍和预处理相关的几个简单内容。

打印出预处理之后的结果

```
$ gcc -E hello.c
```

这样就可以看到源代码中的各种预处理命令是如何被解释的，从而方便理解和查错。

实际上 `gcc` 在这里调用了 `cpp` （虽然通过 `gcc -v` 仅看到 `cc1` ），`cpp` 即 The C Preprocessor，主要用来预处理宏定义、文件包含、条件编译等。下面介绍它的一个比较重要的选项 `-D` 。

在命令行定义宏

```
$ gcc -Dmacro hello.c
```

这个等同于在文件的开头定义宏，即 `#define macro` ，但是在命令行定义更灵活。例如，在源代码中有这些语句。

```
#ifdef DEBUG
printf("this code is for debugging\n");
#endif
```

如果编译时加上 `-DDEBUG` 选项，那么编译器就会把 `printf` 所在的行编译进目标代码，从而方便地跟踪该位置的某些程序状态。这样 `-DDEBUG` 就可以当作一个调试开关，编译时加上它就可以用来打印调试信息，发布时则可以通过去掉该编译选项把调试信息去掉。

编译（翻译）

简述

编译之前，C 语言编译器会进行词法分析、语法分析，接着会把源代码翻译成中间语言，即汇编语言。如果想看到这个中间结果，可以用 `gcc -S` 。需要提到的是，诸如 Shell 等解释语言也会经历一个词法分析和语法分析的阶段，不过之后并不会进行“翻译”，而是“解释”，边解释边执行。

把源代码翻译成汇编语言，实际上是编译的整个过程中的第一个阶段，之后的阶段和汇编语言的开发过程没有什么区别。这个阶段涉及到对源代码的词法分析、语法检查（通过 `-std` 指定遵循哪个标准），并根据优化（ `-O` ）要求进行翻译成汇

编语言的动作。

语法检查

如果仅仅希望进行语法检查，可以用 `gcc` 的 `-fsyntax-only` 选项；如果为了使代码有比较好的可移植性，避免使用 `gcc` 的一些扩展特性，可以结合 `-std` 和 `-pedantic` （或者 `-pedantic-erros` ）选项让源代码遵循某个 C 语言标准的语法。这里演示一个简单的例子：

```
$ cat hello.c
#include <stdio.h>
int main()
{
    printf("hello, world\n")
    return 0;
}
$ gcc -fsyntax-only hello.c
hello.c: In function ‘main’:
hello.c:5: error: expected ‘;’ before ‘return’
$ vim hello.c
$ cat hello.c
#include <stdio.h>
int main()
{
        printf("hello, world\n");
        int i;
        return 0;
}
$ gcc -std=c89 -pedantic-errors hello.c    #默认情况下，gcc是允许在程序中间声明变量的，但是turboc就不支持
hello.c: In function ‘main’:
hello.c:5: error: ISO C90 forbids mixed declarations and code
```

语法错误是程序开发过程中难以避免的错误（人的大脑在很多情况下都容易开小差），不过编译器往往能够通过语法检查快速发现这些错误，并准确地告知语法错误的大概位置。因此，作为开发人员，要做的事情不是“恐慌”（不知所措），而是

认真阅读编译器的提示，根据平时积累的经验（最好总结一份常见语法错误索引，很多资料都提供了常见语法错误列表，如《C Traps & Pitfalls》和编辑器提供的语法检查功能（语法加亮、括号匹配提示等）快速定位语法出错的位置并进行修改。

编译器优化

语法检查之后就是翻译动作， `gcc` 提供了一个优化选项 `-O` ，以便根据不同的运行平台和用户要求产生经过优化的汇编代码。例如，

```
$ gcc -o hello hello.c         # 采用默认选项，不优化
$ gcc -O2 -o hello2 hello.c    # 优化等次是2
$ gcc -Os -o hellos hello.c    # 优化目标代码的大小
$ ls -S hello hello2 hellos    # 可以看到，hellos 比较小, hello2 比较大
hello2  hello  hellos
$ time ./hello
hello, world

real    0m0.001s
user    0m0.000s
sys     0m0.000s
$ time ./hello2     # 可能是代码比较少的缘故，执行效率看上去不是很明显
hello, world

real    0m0.001s
user    0m0.000s
sys     0m0.000s

$ time ./hellos     # 虽然目标代码小了，但是执行效率慢了些
hello, world

real    0m0.002s
user    0m0.000s
sys     0m0.000s
```

根据上面的简单演示，可以看出 `gcc` 有很多不同的优化选项，主要看用户的需求了，目标代码的大小和效率之间貌似存在一个“纠缠”，需要开发人员自己权衡。

生成汇编语言文件

下面通过 `-S` 选项来看看编译出来的中间结果：汇编语言，还是以之前那个 `hello.c` 为例。

```
$ gcc -S hello.c  # 默认输出是hello.s，可自己指定，输出到屏幕`-o -`，
输出到其他文件`-o file`
$ cat hello.s
cat hello.s
        .file   "hello.c"
        .section        .rodata
.LC0:
        .string "hello, world"
        .text
.globl main
        .type   main, @function
main:
        leal    4(%esp), %ecx
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        subl    $4, %esp
        movl    $.LC0, (%esp)
        call    puts
        movl    $0, %eax
        addl    $4, %esp
        popl    %ecx
        popl    %ebp
        leal    -4(%ecx), %esp
        ret
        .size   main, .-main
        .ident  "GCC: (GNU) 4.1.3 20070929 (prerelease) (Ubuntu
4.1.2-16ubuntu2)"
        .section        .note.GNU-stack,"",@progbits
```

不知道看出来没？和课堂里学的 intel 的汇编语法不太一样，这里用的是 `AT&T` 语法格式。如果想学习 Linux 下的汇编语言开发，下一节开始的所有章节基本上覆盖了 Linux 下汇编语言开发的一般过程，不过这里不介绍汇编语言语法。

在学习后面的章节之前，建议自学旧金山大学的微机编程课程 CS 630，该课深入介绍了 Linux/X86 平台下的 `AT&T` 汇编语言开发。如果想在 `Qemu` 上做这个课程里的实验，可以阅读本文作者写的 CS630: Linux 下通过 Qemu 学习 X86 AT&T 汇编语言。

需要补充的是，在写 C 语言代码时，如果能够对编译器比较熟悉（工作原理和一些细节）的话，可能会很有帮助。包括这里的优化选项（有些优化选项可能在汇编时采用）和可能的优化措施，例如字节对齐、条件分支语句裁减（删除一些明显分支）等。

汇编

简述

汇编实际上还是翻译过程，只不过把作为中间结果的汇编代码翻译成了机器代码，即目标代码，不过它还不可以运行。如果要产生这一中间结果，可用 `gcc -c` ，当然，也可通过 `as` 命令处理汇编语言源文件来产生。

汇编是把汇编语言翻译成目标代码的过程，如果有在 Windows 下学习过汇编语言开发，大家应该比较熟悉 `nasm` 汇编工具(支持 Intel 格式的汇编语言)，不过这里主要用 `as` 汇编工具来汇编 `AT&T` 格式的汇编语言，因为 `gcc` 产生的中间代码就是 `AT&T` 格式的。

生成目标代码

下面来演示分别通过 `gcc -c` 选项和 `as` 来产生目标代码。

```
$ file hello.s
hello.s: ASCII assembler program text
$ gcc -c hello.s   #用gcc把汇编语言编译成目标代码
$ file hello.o     #file命令用来查看文件类型，目标代码可重定位的(reloca
table)，
                   #需要通过ld进行进一步链接成可执行程序(executable)和
共享库(shared)
hello.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYS
V), not stripped
$ as -o hello.o hello.s        #用as把汇编语言编译成目标代码
$ file hello.o
hello.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYS
V), not stripped
```

`gcc` 和 `as` 默认产生的目标代码都是 ELF 格式的，因此这里主要讨论ELF格式的目标代码（如果有时间再回顾一下 `a.out` 和 `coff` 格式，当然也可以先了解一下，并结合 `objcopy` 来转换它们，比较异同)。

ELF 文件初次接触

目标代码不再是普通的文本格式，无法直接通过文本编辑器浏览，需要一些专门的工具。如果想了解更多目标代码的细节，区分 `relocatable` （可重定位）、 `executable` （可执行）、 `shared libarary` （共享库）的不同，我们得设法了解目标代码的组织方式和相关的阅读和分析工具。下面主要介绍这部分内容。

> BFD is a package which allows applications to use the same routines to operate on object files whatever the object file format. A new object file format can be supported simply by creating a new BFD back end and adding it to the library.

binutils（GNU Binary Utilities）的很多工具都采用这个库来操作目标文件，这类工具有 `objdump` ， `objcopy` ， `nm` ， `strip` 等（当然，我们也可以利用它。如果深入了解ELF格式，那么通过它来分析和编写 Virus 程序将会更加方便），不过另外一款非常优秀的分析工具 `readelf` 并不是基于这个库，所以也应该可以直接用 `elf.h` 头文件中定义的相关结构来操作 ELF 文件。

下面将通过这些辅助工具（主要是 `readelf` 和 `objdump` ），结合 ELF 手册来分析它们。将依次介绍 ELF 文件的结构和三种不同类型 ELF 文件的区别。

ELF 文件的结构

```
ELF Header(ELF文件头)
Program Headers Table(程序头表，实际上叫段表好一些，用于描述可执行文件和可共享库)
Section 1
Section 2
Section 3
...
Section Headers Table(节区头部表，用于链接可重定位文件成可执行文件或共享库)
```

对于可重定位文件，程序头是可选的，而对于可执行文件和共享库文件（动态链接库），节区表则是可选的。可以分别通过 `readelf` 文件的 `-h` ， `-l` 和 `-S` 参数查看 ELF 文件头（ELF Header）、程序头部表（Program Headers Table，段表）和节区表（Section Headers Table）。

文件头说明了文件的类型，大小，运行平台，节区数目等。

三种不同类型 ELF 文件比较

先来通过文件头看看不同ELF的类型。为了说明问题，先来几段代码吧。

```
/* myprintf.c */
#include <stdio.h>

void myprintf(void)
{
    printf("hello, world!\n");
}
```

```
/* test.h -- myprintf function declaration */

#ifndef _TEST_H_
#define _TEST_H_

void myprintf(void);

#endif
```

```
/* test.c */
#include "test.h"

int main()
{
    myprintf();
    return 0;
}
```

下面通过这几段代码来演示通过 `readelf -h` 参数查看 ELF 的不同类型。期间将演示如何创建动态链接库（即可共享文件）、静态链接库，并比较它们的异同。

编译产生两个目标文件 `myprintf.o` 和 `test.o`，它们都是可重定位文件（REL）：

```
$ gcc -c myprintf.c test.c
$ readelf -h test.o | grep Type
  Type:                              REL (Relocatable file)
$ readelf -h myprintf.o | grep Type
  Type:                              REL (Relocatable file)
```

根据目标代码链接产生可执行文件，这里的文件类型是可执行的(EXEC)：

```
$ gcc -o test myprintf.o test.o
$ readelf -h test | grep Type
  Type:                              EXEC (Executable file)
```

用 `ar` 命令创建一个静态链接库，静态链接库也是可重定位文件（REL）：

```
$ ar rcsv libmyprintf.a myprintf.o
$ readelf -h libmyprintf.a | grep Type
  Type:                              REL (Relocatable file)
```

可见，静态链接库和可重定位文件类型一样，它们之间唯一不同是前者可以是多个可重定位文件的“集合”。

静态链接库可直接链接（只需库名，不要前面的 `lib` ），也可用 `-l` 参数， `-L` 指定库搜索路径。

```
$ gcc -o test test.o -lmyprintf -L./
```

编译产生动态链接库，并支持 `major` 和 `minor` 版本号，动态链接库类型为 `DYN` ：

```
$ gcc -Wall myprintf.o -shared -Wl,-soname,libmyprintf.so.0 -o l
ibmyprintf.so.0.0
$ ln -sf libmyprintf.so.0.0 libmyprintf.so.0
$ ln -sf libmyprintf.so.0 libmyprintf.so
$ readelf -h libmyprintf.so | grep Type
  Type:                              DYN (Shared object file)
```

动态链接库编译时和静态链接库类似：

```
$ gcc -o test test.o -lmyprintf -L./
```

但是执行时需要指定动态链接库的搜索路径，把 `LD_LIBRARY_PATH` 设为当前目录，指定 `test` 运行时的动态链接库搜索路径：

```
$ LD_LIBRARY_PATH=./ ./test
$ gcc -static -o test test.o -lmyprintf -L./
```

在不指定 `-static` 时会优先使用动态链接库，指定时则阻止使用动态链接库，这时会把所有静态链接库文件加入到可执行文件中，使得执行文件很大，而且加载到内存以后会浪费内存空间，因此不建议这么做。

经过上面的演示基本可以看出它们之间的不同：

- 可重定位文件本身不可以运行，仅仅是作为可执行文件、静态链接库（也是可重定位文件）、动态链接库的 “组件”。
- 静态链接库和动态链接库本身也不可以执行，作为可执行文件的“组件”，它们两者也不同，前者也是可重定位文件（只不过可能是多个可重定位文件的集合），并且在链接时加入到可执行文件中去。
- 而动态链接库在链接时，库文件本身并没有添加到可执行文件中，只是在可执行文件中加入了该库的名字等信息，以便在可执行文件运行过程中引用库中的函数时由动态链接器去查找相关函数的地址，并调用它们。

从这个意义上说，动态链接库本身也具有可重定位的特征，含有可重定位的信息。对于什么是重定位？如何进行静态符号和动态符号的重定位，我们将在链接部分和《动态符号链接的细节》一节介绍。

ELF 主体：节区

下面来看看 ELF 文件的主体内容：节区（Section)。

ELF 文件具有很大的灵活性，它通过文件头组织整个文件的总体结构，通过节区表 (Section Headers Table）和程序头（Program Headers Table 或者叫段表）来分别描述可重定位文件和可执行文件。但不管是哪种类型，它们都需要它们的主体，即各种节区。

在可重定位文件中，节区表描述的就是各种节区本身；而在可执行文件中，程序头描述的是由各个节区组成的段（Segment），以便程序运行时动态装载器知道如何对它们进行内存映像，从而方便程序加载和运行。

下面先来看看一些常见的节区，而关于这些节区（Section）如何通过重定位构成不同的段（Segments），以及有哪些常规的段，我们将在链接部分进一步介绍。

可以通过 `readelf -S` 查看 ELF 的节区。（建议一边操作一边看文档，以便加深对 ELF 文件结构的理解）先来看看可重定位文件的节区信息，通过节区表来查看：

默认编译好 `myprintf.c` ，将产生一个可重定位的文件 `myprintf.o` ，这里通过 `myprintf.o` 的节区表查看节区信息。

```
$ gcc -c myprintf.c
$ readelf -S myprintf.o
There are 11 section headers, starting at offset 0xc0:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size
ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000
00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 000018
00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000334 000010
08      9   1  4
  [ 3] .data             PROGBITS        00000000 00004c 000000
00  WA  0   0  4
  [ 4] .bss              NOBITS          00000000 00004c 000000
00  WA  0   0  4
  [ 5] .rodata           PROGBITS        00000000 00004c 00000e
00   A  0   0  1
  [ 6] .comment          PROGBITS        00000000 00005a 000012
00      0   0  1
  [ 7] .note.GNU-stack   PROGBITS        00000000 00006c 000000
00      0   0  1
  [ 8] .shstrtab         STRTAB          00000000 00006c 000051
00      0   0  1
  [ 9] .symtab           SYMTAB          00000000 000278 0000a0
10     10   8  4
  [10] .strtab           STRTAB          00000000 000318 00001a
00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor
 specific)
```

用 `objdump -d` 可看反编译结果，用 `-j` 选项可指定需要查看的节区：

```
$ objdump -d -j .text   myprintf.o
myprintf.o:     file format elf32-i386

Disassembly of section .text:

00000000 <myprintf>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 08                sub    $0x8,%esp
   6:   83 ec 0c                sub    $0xc,%esp
   9:   68 00 00 00 00          push   $0x0
   e:   e8 fc ff ff ff          call   f <myprintf+0xf>
  13:   83 c4 10                add    $0x10,%esp
  16:   c9                      leave
  17:   c3                      ret
```

用 `-r` 选项可以看到有关重定位的信息，这里有两部分需要重定位：

```
$ readelf -r myprintf.o

Relocation section '.rel.text' at offset 0x334 contains 2 entrie
s:
 Offset     Info    Type            Sym.Value  Sym. Name
0000000a  00000501 R_386_32          00000000   .rodata
0000000f  00000902 R_386_PC32        00000000   puts
```

`.rodata` 节区包含只读数据，即我们要打印的 `hello, world!`

```
$ readelf -x .rodata myprintf.o

Hex dump of section '.rodata':
  0x00000000 68656c6c 6f2c2077 6f726c64 2100     hello, world!.
```

没有找到 `.data` 节区，它应该包含一些初始化的数据：

```
$ readelf -x .data myprintf.o

Section '.data' has no data to dump.
```

也没有 `.bss` 节区，它应该包含一些未初始化的数据，程序默认初始为 0：

```
$ readelf -x .bss       myprintf.o

Section '.bss' has no data to dump.
```

`.comment` 是一些注释，可以看到是是 `Gcc` 的版本信息

```
$ readelf -x .comment myprintf.o

Hex dump of section '.comment':
  0x00000000 00474343 3a202847 4e552920 342e312e .GCC: (GNU) 4.1
.
  0x00000010 3200                                2.
```

`.note.GNU-stack` 这个节区也没有内容：

```
$ readelf -x .note.GNU-stack myprintf.o

Section '.note.GNU-stack' has no data to dump.
```

`.shstrtab` 包括所有节区的名字：

```
$ readelf -x .shstrtab myprintf.o

Hex dump of section '.shstrtab':
  0x00000000 002e7379 6d746162 002e7374 72746162 ..symtab..strta
b
  0x00000010 002e7368 73747274 6162002e 72656c2e ..shstrtab..rel
.
  0x00000020 74657874 002e6461 7461002e 62737300 text..data..bss
.
  0x00000030 2e726f64 61746100 2e636f6d 6d656e74 .rodata..commen
t
  0x00000040 002e6e6f 74652e47 4e552d73 7461636b ..note.GNU-stac
k
  0x00000050 00                                  .
```

符号表 `.symtab` 包括所有用到的相关符号信息，如函数名、变量名，可用 `readelf` 查看：

```
$ readelf -symtab myprintf.o

Symbol table '.symtab' contains 10 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS myprintf.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    7
     7: 00000000     0 SECTION LOCAL  DEFAULT    6
     8: 00000000    24 FUNC    GLOBAL DEFAULT    1 myprintf
     9: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
```

字符串表 `.strtab` 包含用到的字符串，包括文件名、函数名、变量名等：

```
$ readelf -x .strtab myprintf.o

Hex dump of section '.strtab':
  0x00000000 006d7970 72696e74 662e6300 6d797072 .myprintf.c.myp
r
  0x00000010 696e7466 00707574 7300              intf.puts.
```

从上表可以看出，对于可重定位文件，会包含这些基本节区 `.text`，`.rel.text`，`.data`，`.bss`，`.rodata`，`.comment`，`.note.GNU-stack`，`.shstrtab`，`.symtab` 和 `.strtab`。

汇编语言文件中的节区表述

为了进一步理解这些节区和源代码的关系，这里来看一看 `myprintf.c` 产生的汇编代码。

```
$ gcc -S myprintf.c
$ cat myprintf.s
        .file   "myprintf.c"
        .section        .rodata
.LC0:
        .string "hello, world!"
        .text
.globl myprintf
        .type   myprintf, @function
myprintf:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        subl    $12, %esp
        pushl   $.LC0
        call    puts
        addl    $16, %esp
        leave
        ret
        .size   myprintf, .-myprintf
        .ident  "GCC: (GNU) 4.1.2"
        .section        .note.GNU-stack,"",@progbits
```

是不是可以从中看出可重定位文件中的那些节区和汇编语言代码之间的关系？在上面的可重定位文件，可以看到有一个可重定位的节区，即 `.rel.text` ，它标记了两个需要重定位的项， `.rodata` 和 `puts` 。这个节区将告诉编译器这两个信息在链接或者动态链接的过程中需要重定位， 具体如何重定位？将根据重定位项的类型，比如上面的 `R_386_32` 和 `R_386_PC32` 。

到这里，对可重定位文件应该有了一个基本的了解，下面将介绍什么是可重定位，可重定位文件到底是如何被链接生成可执行文件和动态链接库的，这个过程除了进行一些符号的重定位外，还进行了哪些工作呢？

链接

简述

重定位是将符号引用与符号定义进行链接的过程。因此链接是处理可重定位文件，把它们的各种符号引用和符号定义转换为可执行文件中的合适信息（一般是虚拟内存地址）的过程。

链接又分为静态链接和动态链接，前者是程序开发阶段程序员用 `ld` （ `gcc` 实际上在后台调用了 `ld` ）静态链接器手动链接的过程，而动态链接则是程序运行期间系统调用动态链接器（ `ld-linux.so` ）自动链接的过程。

比如，如果链接到可执行文件中的是静态链接库 `libmyprintf.a` ，那么 `.rodata` 节区在链接后需要被重定位到一个绝对的虚拟内存地址，以便程序运行时能够正确访问该节区中的字符串信息。而对于 `puts` 函数，因为它是动态链接库 `libc.so` 中定义的函数，所以会在程序运行时通过动态符号链接找出 `puts` 函数在内存中的地址，以便程序调用该函数。在这里主要讨论静态链接过程，动态链接过程见《动态符号链接的细节》。

静态链接过程主要是把可重定位文件依次读入，分析各个文件的文件头，进而依次读入各个文件的节区，并计算各个节区的虚拟内存位置，对一些需要重定位的符号进行处理，设定它们的虚拟内存地址等，并最终产生一个可执行文件或者是动态链接库。这个链接过程是通过 `ld` 来完成的， `ld` 在链接时使用了一个链接脚本（ `linker script` ），该链接脚本处理链接的具体细节。

由于静态符号链接过程非常复杂，特别是计算符号地址的过程，考虑到时间关系，相关细节请参考 ELF 手册。这里主要介绍可重定位文件中的节区（节区表描述的）和可执行文件中段（程序头描述的）的对应关系以及 `gcc` 编译时采用的一些默认链接选项。

可执行文件的段：节区重排

下面先来看看可执行文件的节区信息，通过程序头（段表）来查看，为了比较，先把 `test.o` 的节区表也列出：

```
$ readelf -S test.o
There are 10 section headers, starting at offset 0xb4:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size
ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000
```

```
00       0   0   0
  [ 1] .text             PROGBITS        00000000 000034 000024
00   AX   0   0   4
  [ 2] .rel.text         REL             00000000 0002ec 000008
08       8   1   4
  [ 3] .data             PROGBITS        00000000 000058 000000
00   WA   0   0   4
  [ 4] .bss              NOBITS          00000000 000058 000000
00   WA   0   0   4
  [ 5] .comment          PROGBITS        00000000 000058 000012
00       0   0   1
  [ 6] .note.GNU-stack   PROGBITS        00000000 00006a 000000
00       0   0   1
  [ 7] .shstrtab         STRTAB          00000000 00006a 000049
00       0   0   1
  [ 8] .symtab           SYMTAB          00000000 000244 000090
10       9   7   4
  [ 9] .strtab           STRTAB          00000000 0002d4 000016
00       0   0   1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor
 specific)
$ gcc -o test test.o myprintf.o
$ readelf -l test

Elf file type is EXEC (Executable file)
Entry point 0x80482b0
There are 7 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz
Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000e0 0x000e0
R E 0x4
  INTERP         0x000114 0x08048114 0x08048114 0x00013 0x00013
R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x0047c 0x0047c
```

```
R E 0x1000
  LOAD           0x00047c 0x0804947c 0x0804947c 0x00104 0x00108
RW  0x1000
  DYNAMIC        0x000490 0x08049490 0x08049490 0x000c8 0x000c8
RW  0x4
  NOTE           0x000128 0x08048128 0x08048128 0x00020 0x00020
R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000
RW  0x4

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.versi
on .gnu.version_r
          .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_f
rame
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag
   06
```

可发现，`test` 和 `test.o`，`myprintf.o` 相比，多了很多节区，如 `.interp` 和 `.init` 等。另外，上表也给出了可执行文件的如下几个段（Segment）：

- `PHDR`：给出了程序表自身的大小和位置，不能出现一次以上。
- `INTERP`：因为程序中调用了 `puts`（在动态链接库中定义），使用了动态链接库，因此需要动态装载器／链接器（`ld-linux.so`）
- `LOAD`：包括程序的指令，`.text` 等节区都映射在该段，只读（R）
- `LOAD`：包括程序的数据，`.data`，`.bss` 等节区都映射在该段，可读写（RW）
- `DYNAMIC`：动态链接相关的信息，比如包含有引用的动态链接库名字等信息
- `NOTE`：给出一些附加信息的位置和大小
- `GNU_STACK`：这里为空，应该是和GNU相关的一些信息

这里的段可能包括之前的一个或者多个节区，也就是说经过链接之后原来的节区被重排了，并映射到了不同的段，这些段将告诉系统应该如何把它加载到内存中。

链接背后的故事

从上表中，通过比较可执行文件 `test` 中拥有的节区和可重定位文件（`test.o` 和 `myprintf.o`）中拥有的节区后发现，链接之后多了一些之前没有的节区，这些新的节区来自哪里？它们的作用是什么呢？先来通过 `gcc -v` 看看它的后台链接过程。

把可重定位文件链接成可执行文件：

```
$ gcc -v -o test test.o myprintf.o
Reading specs from /usr/lib/gcc/i486-slackware-linux/4.1.2/specs
Target: i486-slackware-linux
Configured with: ../gcc-4.1.2/configure --prefix=/usr --enable-s
hared
--enable-languages=ada,c,c++,fortran,java,objc --enable-threads=
posix
--enable-__cxa_atexit --disable-checking --with-gnu-ld --verbose
--with-arch=i486 --target=i486-slackware-linux --host=i486-slack
ware-linux
Thread model: posix
gcc version 4.1.2
 /usr/libexec/gcc/i486-slackware-linux/4.1.2/collect2 --eh-frame
-hdr -m
elf_i386 -dynamic-linker /lib/ld-linux.so.2 -o test
/usr/lib/gcc/i486-slackware-linux/4.1.2/../../../crt1.o
/usr/lib/gcc/i486-slackware-linux/4.1.2/../../../crti.o
/usr/lib/gcc/i486-slackware-linux/4.1.2/crtbegin.o
-L/usr/lib/gcc/i486-slackware-linux/4.1.2
-L/usr/lib/gcc/i486-slackware-linux/4.1.2
-L/usr/lib/gcc/i486-slackware-linux/4.1.2/../../../../i486-slack
ware-linux/lib
-L/usr/lib/gcc/i486-slackware-linux/4.1.2/../../.. test.o myprin
tf.o -lgcc
--as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s
 --no-as-needed
/usr/lib/gcc/i486-slackware-linux/4.1.2/crtend.o
/usr/lib/gcc/i486-slackware-linux/4.1.2/../../../crtn.o
```

从上述演示看出，`gcc` 在链接了我们自己的目标文件 `test.o` 和 `myprintf.o` 之外，还链接了 `crt1.o`，`crtbegin.o` 等额外的目标文件，难道那些新的节区就来自这些文件？

用 ld 完成链接过程

另外 `gcc` 在进行了相关配置（`./configure`）后，调用了 `collect2`，却并没有调用 `ld`，通过查找 `gcc` 文档中和 `collect2` 相关的部分发现 `collect2` 在后台实际上还是去寻找 `ld` 命令的。为了理解 `gcc` 默认链接的后台细节，这里直接把 `collect2` 替换成 `ld`，并把一些路径换成绝对路径或者简化，得到如下的 `ld` 命令以及执行的效果。

```
$ ld --eh-frame-hdr \
-m elf_i386 \
-dynamic-linker /lib/ld-linux.so.2 \
-o test \
/usr/lib/crt1.o /usr/lib/crti.o /usr/lib/gcc/i486-slackware-linu
x/4.1.2/crtbegin.o \
test.o myprintf.o \
-L/usr/lib/gcc/i486-slackware-linux/4.1.2 -L/usr/i486-slackware-
linux/lib -L/usr/lib/ \
-lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -
lgcc_s --no-as-needed \
/usr/lib/gcc/i486-slackware-linux/4.1.2/crtend.o /usr/lib/crtn.o
$ ./test
hello, world!
```

不出所料，它完美地运行了。下面通过 `ld` 的手册（`man ld`）来分析一下这几个参数：

- `--eh-frame-hdr`

  要求创建一个 `.eh_frame_hdr` 节区(貌似目标文件test中并没有这个节区，所以不关心它)。

- `-m elf_i386`

这里指定不同平台上的链接脚本，可以通过 `--verbose` 命令查看脚本的具体内容，如 `ld -m elf_i386 --verbose` ，它实际上被存放在一个文件中（ `/usr/lib/ldscripts` 目录下），我们可以去修改这个脚本，具体如何做？请参考 `ld` 的手册。在后面我们将简要提到链接脚本中是如何预定义变量的，以及这些预定义变量如何在我们的程序中使用。需要提到的是，如果不是交叉编译，那么无须指定该选项。

- -dynamic-linker /lib/ld-linux.so.2

  指定动态装载器/链接器，即程序中的 `INTERP` 段中的内容。动态装载器/链接器负责链接有可共享库的可执行文件的装载和动态符号链接。

- -o test

  指定输出文件，即可执行文件名的名字

- /usr/lib/crt1.o /usr/lib/crti.o /usr/lib/gcc/i486-slackware-linux/4.1.2/crtbegin.o

  链接到 `test` 文件开头的一些内容，这里实际上就包含了 `.init` 等节区。 `.init` 节区包含一些可执行代码，在 `main` 函数之前被调用，以便进行一些初始化操作，在 C++ 中完成构造函数功能。

- test.o myprintf.o

  链接我们自己的可重定位文件

- `-L/usr/lib/gcc/i486-slackware-linux/4.1.2 -L/usr/i486-slackware-linux/lib -L/usr/lib/ \ -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed`

  链接 `libgcc` 库和 `libc` 库，后者定义有我们需要的 `puts` 函数

- /usr/lib/gcc/i486-slackware-linux/4.1.2/crtend.o /usr/lib/crtn.o

  链接到 `test` 文件末尾的一些内容，这里实际上包含了 `.fini` 等节区。 `.fini` 节区包含了一些可执行代码，在程序退出时被执行，作一些清理工作，在 C++ 中完成析构造函数功能。我们往往可以通过 `atexit` 来注册那些需要在程序退出时才执行的函数。

C++构造与析构：crtbegin.o和crtend.o

对于 `crtbegin.o` 和 `crtend.o` 这两个文件，貌似完全是用来支持 C++ 的构造和析构工作的，所以可以不链接到我们的可执行文件中，链接时把它们去掉看看，

```
$ ld -m elf_i386 -dynamic-linker /lib/ld-linux.so.2 -o test \
  /usr/lib/crt1.o /usr/lib/crti.o test.o myprintf.o \
  -L/usr/lib -lc /usr/lib/crtn.o    #后面发现不用链接libgcc，也不用-
-eh-frame-hdr参数
$ readelf -l test

Elf file type is EXEC (Executable file)
Entry point 0x80482b0
There are 7 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz
Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000e0 0x000e0
R E 0x4
  INTERP         0x000114 0x08048114 0x08048114 0x00013 0x00013
R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x003ea 0x003ea
R E 0x1000
  LOAD           0x0003ec 0x080493ec 0x080493ec 0x000e8 0x000e8
RW  0x1000
  DYNAMIC        0x0003ec 0x080493ec 0x080493ec 0x000c8 0x000c8
RW  0x4
  NOTE           0x000128 0x08048128 0x08048128 0x00020 0x00020
R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000
RW  0x4

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.versi
on .gnu.version_r
          .rel.dyn .rel.plt .init .plt .text .fini .rodata
   03     .dynamic .got .got.plt .data
```

```
    04     .dynamic
    05     .note.ABI-tag
    06
$ ./test
hello, world!
```

完全可以工作，而且发现 `.ctors` （保存着程序中全局构造函数的指针数组）， `.dtors` （保存着程序中全局析构函数的指针数组）， `.jcr` （未知）， `.eh_frame` 节区都没有了，所以 `crtbegin.o` 和 `crtend.o` 应该包含了这些节区。

初始化与退出清理：crti.o 和 crtn.o

而对于另外两个文件 `crti.o` 和 `crtn.o` ，通过 `readelf -S` 查看后发现它们都有 `.init` 和 `.fini` 节区，如果我们不需要让程序进行一些初始化和清理工作呢？是不是就可以不链接这个两个文件？试试看。

```
$ ld  -m elf_i386 -dynamic-linker /lib/ld-linux.so.2 -o test \
      /usr/lib/crt1.o test.o myprintf.o -L/usr/lib/ -lc
/usr/lib/libc_nonshared.a(elf-init.oS): In function `__libc_csu_
init':
(.text+0x25): undefined reference to `_init'
```

貌似不行，竟然有人调用了 `__libc_csu_init` 函数，而这个函数引用了 `_init` 。这两个符号都在哪里呢？

```
$ readelf -s /usr/lib/crt1.o | grep __libc_csu_init
    18: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND __libc_csu_in
it
$ readelf -s /usr/lib/crti.o | grep _init
    17: 00000000     0 FUNC    GLOBAL DEFAULT    5 _init
```

竟然是 `crt1.o` 调用了 `__libc_csu_init` 函数，而该函数却引用了我们没有链接的 `crti.o` 文件中定义的 `_init` 符号。这样的话不链接 `crti.o` 和 `crtn.o` 文件就不成了罗？不对吧，要不干脆不用 `crt1.o` 算了，看看 `gcc` 额外链接进去的最后一个文件 `crt1.o` 到底干了个啥子？

```
$ ld  -m elf_i386 -dynamic-linker /lib/ld-linux.so.2 -o \
      test test.o myprintf.o -L/usr/lib/ -lc
ld: warning: cannot find entry symbol _start; defaulting to 0000
0000080481a4
```

这样却说没有找到入口符号 `_start`，难道 `crt1.o` 中定义了这个符号？不过它给默认设置了一个地址，只是个警告，说明 `test` 已经生成，不管怎样先运行看看再说。

```
$ ./test
hello, world!
Segmentation fault
```

貌似程序运行完了，不过结束时冒出个段错误？可能是程序结束时有问题，用 `gdb` 调试看看：

```
$ gcc -g -c test.c myprintf.c #产生目标代码, 非交叉编译，不指定-m也可链接，所以下面可去掉-m
$ ld -dynamic-linker /lib/ld-linux.so.2 -o test \
     test.o myprintf.o -L/usr/lib -lc
ld: warning: cannot find entry symbol _start; defaulting to 00000000080481d8
$ ./test
hello, world!
Segmentation fault
$ gdb -q ./test
(gdb) l
1       #include "test.h"
2
3       int main()
4       {
5               myprintf();
6               return 0;
7       }
(gdb) break 7      #在程序的末尾设置一个断点
Breakpoint 1 at 0x80481bf: file test.c, line 7.
(gdb) r            #程序都快结束了都没问题，怎么会到最后出个问题呢？
Starting program: /mnt/hda8/Temp/c/program/test
hello, world!

Breakpoint 1, main () at test.c:7
7       }
(gdb) n        #单步执行看看，怎么下面一条指令是0x00000001，肯定是程序退出以后出了问题
0x00000001 in ?? ()
(gdb) n        #诶，当然找不到边了，都跑到0x00000001了
Cannot find bounds of current function
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x00000001 in ?? ()
```

原来是这么回事，估计是 `return 0` 返回之后出问题了，看看它的汇编去。

```
$ gcc -S test.c #产生汇编代码
$ cat test.s
...
        call    myprintf
        movl    $0, %eax
        addl    $4, %esp
        popl    %ecx
        popl    %ebp
        leal    -4(%ecx), %esp
        ret
...
```

后面就这么几条指令，难不成 `ret` 返回有问题，不让它 `ret` 返回，把 `return` 改成 `_exit` 直接进入内核退出。

```
$ vim test.c
$ cat test.c    #就把return语句修改成_exit了。
#include "test.h"
#include <unistd.h> /* _exit */

int main()
{
    myprintf();
    _exit(0);
}
$ gcc -g -c test.c myprintf.c
$ ld -dynamic-linker /lib/ld-linux.so.2 -o test test.o myprintf.
o -L/usr/lib -lc
ld: warning: cannot find entry symbol _start; defaulting to 0000
0000080481d8
$ ./test    #竟然好了，再看看汇编有什么不同
hello, world!
$ gcc -S test.c
$ cat test.s    #貌似就把ret指令替换成了_exit函数调用，直接进入内核，让
内核处理了，那为什么ret有问题呢？
...
        call    myprintf
        subl    $12, %esp
        pushl   $0
```

```
        call    _exit
...
$ gdb -q ./test    #把代码改回去（改成return 0;），再调试看看调用main
函数返回时的下一条指令地址eip
(gdb) l
warning: Source file is more recent than executable.
1       #include "test.h"
2
3       int main()
4       {
5               myprintf();
6               return 0;
7       }
(gdb) break 5
Breakpoint 1 at 0x80481b5: file test.c, line 5.
(gdb) break 7
Breakpoint 2 at 0x80481bc: file test.c, line 7.
(gdb) r
Starting program: /mnt/hda8/Temp/c/program/test

Breakpoint 1, main () at test.c:5
5               myprintf();
(gdb) x/8x $esp
0xbf929510:     0xbf92953c      0x080481a4      0x00000000
0xb7eea84f
0xbf929520:     0xbf92953c      0xbf929534      0x00000000
0x00000001
```

发现 `0x00000001` 刚好是之前调试时看到的程序返回后的位置，即 `eip` ，说明程序在初始化时，这个 `eip` 就是错误的。为什么呢？因为根本没有链接进初始化的代码，而是在编译器自己给我们，初始化了程序入口即 `00000000080481d8，也就是说，没有人调用` main ，main `不知道返回哪里去，所以，我们直接让` main `结束时进入内核调用` _exit` 而退出则不会有问题。

通过上面的演示和解释发现只要把return语句修改为_exit语句，程序即使不链接任何额外的目标代码都可以正常运行（原因是不链接那些额外的文件时相当于没有进行初始化操作，如果在程序的最后执行ret汇编指令，程序将无法获得正确的eip，从而无法进行后续的动作）。但是为什么会有“找不到_start符号”的警告呢？通

过 `readelf -s` 查看crt1.o发现里头有这个符号，并且crt1.o引用了main这个符号，是不是意味着会从 `_start` 进入main呢？是不是程序入口是 `_start` ，而并非main呢？

C 语言程序真正的入口

先来看看刚才提到的链接器的默认链接脚本（ `ld -m elf_386 --verbose` ），它告诉我们程序的入口（entry）是 `_start` ，而一个可执行文件必须有一个入口地址才能运行，所以这就是说明了为什么 `ld` 一定要提示我们 “_start找不到”，找不到以后就给默认设置了一个地址。

```
$ ld --verbose  | grep ^ENTRY    #非交叉编译，可不用-m参数；ld默认找_start入口，并不是main哦！
ENTRY(_start)
```

原来是这样，程序的入口（entry）竟然不是 `main` 函数，而是 `_start` 。那干脆把汇编里头的 `main` 给改掉算了，看行不行？

先生成汇编 `test.s` ：

```
$ cat test.c
#include "test.h"
#include <unistd.h>     /* _exit */

int main()
{
    myprintf();
    _exit(0);
}
$ gcc -S test.c
```

然后把汇编中的 `main` 改为 `_start` ，即改程序入口为 `_start` ：

```
$ sed -i -e "s#main#_start#g" test.s
$ gcc -c test.s myprintf.c
```

重新链接，发现果然没问题了：

```
$ ld -dynamic-linker /lib/ld-linux.so.2 -o test test.o myprintf.
o -L/usr/lib/ -lc
$ ./test
hello, world!
```

`_start` 竟然是真正的程序入口，那在有 `main` 的情况下呢？为什么在 `_start` 之后能够找到 `main` 呢？这个看看 alert7 大叔的Before main 分析吧，这里不再深入介绍。

总之呢，通过修改程序的 `return` 语句为 `_exit(0)` 和修改程序的入口为 `_start`，我们的代码不链接 `gcc` 默认链接的那些额外的文件同样可以工作得很好。并且打破了一个学习 C 语言以来的常识：`main` 函数作为程序的主函数，是程序的入口，实际上则不然。

链接脚本初次接触

再补充一点内容，在 `ld` 的链接脚本中，有一个特别的关键字 `PROVIDE`，由这个关键字定义的符号是 `ld` 的预定义字符，我们可以在 C 语言函数中扩展它们后直接使用。这些特别的符号可以通过下面的方法获取，

```
$ ld --verbose | grep PROVIDE | grep -v HIDDEN
  PROVIDE (__executable_start = 0x08048000); . = 0x08048000 + SI
ZEOF_HEADERS;
  PROVIDE (__etext = .);
  PROVIDE (_etext = .);
  PROVIDE (etext = .);
  _edata = .; PROVIDE (edata = .);
  _end = .; PROVIDE (end = .);
```

这里面有几个我们比较关心的，第一个是程序的入口地址 `__executable_start`，另外三个是 `etext`，`edata`，`end`，分别对应程序的代码段（text）、初始化数据（data）和未初始化的数据（bss）（可参考 `man etext`），如何引用这些变量呢？看看这个例子。

```
/* predefinevalue.c */
#include <stdio.h>

extern int __executable_start, etext, edata, end;

int main(void)
{
    printf ("program entry: 0x%x \n", &__executable_start);
    printf ("etext address(text segment): 0x%x \n", &etext);
    printf ("edata address(initilized data): 0x%x \n", &edata);
    printf ("end address(uninitilized data): 0x%x \n", &end);

    return 0;
}
```

到这里，程序链接过程的一些细节都介绍得差不多了。在《动态符号链接的细节》中将主要介绍 ELF 文件的动态符号链接过程。

参考资料

- Linux 汇编语言开发指南
- PowerPC 汇编
- 用于 Power 体系结构的汇编语言
- Linux 中 x86 的内联汇编
- Linux Assembly HOWTO
- Linux Assembly Language Programming
- Guide to Assembly Language Programming in Linux
- An beginners guide to compiling programs under Linux
- gcc manual
- A Quick Tour of Compiling, Linking, Loading, and Handling Libraries on Unix
- Unix 目标文件初探
- Before main()分析
- A Process Viewing Its Own /proc//map Information
- UNIX 环境高级编程
- Linux Kernel Primer
- Understanding ELF using readelf and objdump

- Study of ELF loading and relocs
- ELF file format and ABI
  - [1]
  - [2]
- TN05.ELF.Format.Summary.pdf
- ELF文件格式(中文)
- 关于 Gcc 方面的论文，请查看历年的会议论文集
  - 2005
  - 2006
- The Linux GCC HOW TO
- ELF: From The Programmer's Perspective
- C/C++ 程序编译步骤详解
- C 语言常见问题集
- 使用 BFD 操作 ELF
- bfd document
- UNIX/LINUX 平台可执行文件格式分析
- Linux 汇编语言快速上手：4大架构一块学
- GNU binutils 小结

程序执行的一刹那

当我们在 Linux 下的命令行输入一个命令之后，这背后发生了什么？

什么是命令行接口

用户使用计算机有两种常见的方式，一种是图形化的接口（GUI），另外一种则是命令行接口（CLI）。对于图形化的接口，用户点击某个图标就可启动后台的某个程序；对于命令行的接口，用户键入某个程序的名字就可启动某个程序。这两者的基本过程是类似的，都需要查找程序文件在磁盘上的位置，加载到内存并通过不同的解释器进行解析和运行。下面以命令行为例来介绍程序执行一刹那发生的一些事情。

首先来介绍什么是命令行？命令行就是 `Command Line` ，很直观的概念就是系统启动后的那个黑屏幕：有一个提示符，并有光标在闪烁的那样一个终端，一般情况下可以用 `CTRL+ALT+F1-6` 切换到不同的终端；在 GUI 界面下也会有一些伪终端，看上去和系统启动时的那个终端没有什么区别，也会有一个提示符，并有一个光标在闪烁。就提示符和响应用户的键盘输入而言，它们两者在功能上是一样的，实际上它们就是同一个东西，用下面的命令就可以把它们打印出来。

```
$ echo $SHELL # 打印当前SHELL，当前运行的命令行接口程序
/bin/bash
$ echo $$     # 该程序对应进程ID，$$是个特殊的环境变量，它存放了当前进程I
D
1481
$ ps -C bash   # 通过PS命令查看
  PID TTY          TIME CMD
 1481 pts/0    00:00:00 bash
```

从上面的操作结果可以看出，当前命令行接口实际上是一个程序，那就是 `/bin/bash`，它是一个实实在在的程序，它打印提示符，接受用户输入的命令，分析命令序列并执行然后返回结果。不过 `/bin/bash` 仅仅是当前使用的命令行程序之一，还有很多具有类似功能的程序，比如 `/bin/ash`, `/bin/dash` 等。不过这里主要来讨论 `bash`，讨论它自己是怎么启动的，它怎么样处理用户输入的命令等后台细节？

/bin/bash 是什么时候启动的

/bin/login

先通过 `CTRL+ALT+F1` 切换到一个普通终端下面，一般情况下看到的是 "XXX login: " 提示输入用户名，接着是提示输入密码，然后呢？就直接登录到了我们的命令行接口。实际上正是你输入正确的密码后，那个程序才把 `/bin/bash` 给启动了。那是什么东西提示 "XXX login:" 的呢？正是 `/bin/login` 程序，那 `/bin/login` 程序怎么知道要启动 `/bin/bash`，而不是其他的 `/bin/dash` 呢？

`/bin/login` 程序实际上会检查我们的 `/etc/passwd` 文件，在这个文件里头包含了用户名、密码和该用户的登录 Shell。密码和用户名匹配用户的登录，而登录 Shell 则作为用户登录后的命令行程序。看看 `/etc/passwd` 中典型的这么一行：

```
$ cat /etc/passwd | grep falcon
falcon:x:1000:1000:falcon,,,:/home/falcon:/bin/bash
```

这个是我用的帐号的相关信息哦，看到最后一行没？ `/bin/bash` ，这正是我登录用的命令行解释程序。至于密码呢，看到那个 `x` 没？这个 `x` 说明我的密码被保存在另外一个文件里头 `/etc/shadow` ，而且密码是经过加密的。至于这两个文件的更多细节，看手册吧。

我们怎么知道刚好是 `/bin/login` 打印了 "XXX login" 呢？现在回顾一下很早以前学习的那个 `strace` 命令。我们可以用 `strace` 命令来跟踪 `/bin/login` 程序的执行。

跟上面一样，切换到一个普通终端，并切换到 Root 用户，用下面的命令：

```
$ strace -f -o strace.out /bin/login
```

退出以后就可以打开 `strace.out` 文件，看看到底执行了哪些文件，读取了哪些文件。从中可以看到正是 `/bin/login` 程序用 `execve` 调用了 `/bin/bash` 命令。通过后面的演示，可以发现 `/bin/login` 只是在子进程里头用 `execve` 调用了 `/bin/bash` ，因为在启动 `/bin/bash` 后，可以看到 `/bin/login` 并没有退出。

/bin/getty

那 `/bin/login` 又是怎么起来的呢？

下面再来看一个演示。先在一个可以登陆的终端下执行下面的命令。

```
$ getty 38400 tty8 linux
```

`getty` 命令停留在那里，貌似等待用户的什么操作，现在切回到第 8 个终端，是不是看到有 "XXX login:" 的提示了。输入用户名并登录，之后退出，回到第一个终端，发现 `getty` 命令已经退出。

类似地，也可以用 `strace` 命令来跟踪 `getty` 的执行过程。在第一个终端下切换到 Root 用户。执行如下命令：

```
$ strace -f -o strace.out getty 38400 tty8 linux
```

同样在 `strace.out` 命令中可以找到该命令的相关启动细节。比如，可以看到正是 `getty` 程序用 `execve` 系统调用执行了 `/bin/login` 程序。这个地方，`getty` 是在自己的主进程里头直接执行了 `/bin/login`，这样 `/bin/login` 将把 `getty` 的进程空间替换掉。

/sbin/init

这里涉及到一个非常重要的东西：`/sbin/init`，通过 `man init` 命令可以查看到该命令的作用，它可是“万物之王”（init is the parent of all processes on the system）哦。它是 Linux 系统默认启动的第一个程序，负责进行 Linux 系统的一些初始化工作，而这些初始化工作的配置则是通过 `/etc/inittab` 来做的。那么来看看 `/etc/inittab` 的一个简单的例子吧，可以通过 `man inittab` 查看相关帮助。

需要注意的是，在较新版本的 Ubuntu 和 Fedora 等发行版中，一些新的 `init` 程序，比如 `upstart` 和 `systemd` 被开发出来用于取代 `System V init`，它们可能放弃了对 `/etc/inittab` 的使用，例如 `upstart` 会读取 `/etc/init/` 下的配置，比如 `/etc/init/tty1.conf`，但是，基本的配置思路还是类似 `/etc/inittab`，对于 `upstart` 的 `init` 配置，这里不做介绍，请通过 `man 5 init` 查看帮助。

配置文件 `/etc/inittab` 的语法非常简单，就是下面一行的重复，

```
id:runlevels:action:process
```

- `id` 就是一个唯一的编号，不用管它，一个名字而言，无关紧要。
- `runlevels` 是运行级别，这个还是比较重要的，理解运行级别的概念很有必要，它可以有如下的取值：

  ```
  0 is halt.
  1 is single-user.
  2-5 are multi-user.
  6 is reboot.
  ```

不过，真正在配置文件里头用的是 `1-5` 了，而 `0` 和 `6` 非常特别，除了用它作为 `init` 命令的参数关机和重启外，似乎没有哪个“傻瓜”把它写在系统的配置文件里头，让系统启动以后就关机或者重启。 `1` 代表单用户，而 `2-5` 则代表多用户。对于 `2-5` 可能有不同的解释，比如在 Slackware 12.0 上， `2,3,5` 被用来作为多用户模式，但是默认不启动 X windows （GUI接口），而 `4` 则作为启动 X windows 的运行级别。

- `action` 是动作，它也有很多选择，我们关心几个常用的
- `initdefault` ：用来指定系统启动后进入的运行级别，通常在 `/etc/inittab` 的第一条配置，如：

```
id:3:initdefault:
```

这个说明默认运行级别是 3，即多用户模式，但是不启动 X window 的那种。

- `sysinit` ：指定那些在系统启动时将被执行的程序，例如：

```
si:S:sysinit:/etc/rc.d/rc.S
```

在 `man inittab` 中提到，对于 `sysinit` ， `boot` 等动作， `runlevels` 选项是不用管的，所以可以很容易解读这条配置：它的意思是系统启动时将默认执行 `/etc/rc.d/rc.S` 文件，在这个文件里可直接或者间接地执行想让系统启动时执行的任何程序，完成系统的初始化。

- `wait` ：当进入某个特别的运行级别时，指定的程序将被执行一次， `init` 将等到它执行完成，例如：

```
rc:2345:wait:/etc/rc.d/rc.M
```

这个说明无论是进入运行级别 2，3，4，5 中哪一个， `/etc/rc.d/rc.M` 将被执行一次，并且有 `init` 等待它执行完成。

- `ctrlaltdel` ，当 `init` 程序接收到 `SIGINT` 信号时，某个指定的程序将被执行，我们通常通过按下 `CTRL+ALT+DEL` ，这个默认情况下将给 `init` 发送一个 `SIGINT` 信号。

如果我们想在按下这几个键时，系统重启，那么可以在 `/etc/inittab` 中写入：

```
ca::ctrlaltdel:/sbin/shutdown -t5 -r now
```

- `respawn`：这个指定的进程将被重启，任何时候当它退出时。这意味着没有办法结束它，除非 `init` 自己结束了。例如：

```
c1:1235:respawn:/sbin/agetty 38400 tty1 linux
```

这一行的意思非常简单，就是系统运行在级别 1，2，3，5 时，将默认执行 `/sbin/agetty` 程序（这个类似于上面提到的 `getty` 程序），这个程序非常有意思，就是无论什么时候它退出，`init` 将再次启动它。这个有几个比较有意思的问题：

- 在 Slackware 12.0 下，当默认运行级别为 4 时，只有第 6 个终端可以用。原因是什么呢？因为类似上面的配置，因为那里只有 `1235`，而没有 `4`，这意味着当系统运行在第 `4` 级别时，其他终端下的 `/sbin/agetty` 没有启动。所以，如果想让其他终端都可以用，把 `1235` 修改为 `12345` 即可。
- 另外一个有趣的问题就是：正是 `init` 程序在读取这个配置行以后启动了 `/sbin/agetty`，这就是 `/sbin/agetty` 的秘密。
- 还有一个问题：无论退出哪个终端，那个 "XXX login:" 总是会被打印，原因是 `respawn` 动作有趣的性质，因为它告诉 `init`，无论 `/sbin/agetty` 什么时候退出，重新把它启动起来，那跟 "XXX login:" 有什么关系呢？从前面的内容，我们发现正是 `/sbin/getty` （同 `agetty`）启动了 `/bin/login`，而 `/bin/login` 又启动了 `/bin/bash`，即我们的命令行程序。

命令启动过程追本溯源

而 `init` 程序作为“万物之王”，它是所有进程的“父”（也可能是祖父……）进程，那意味着其他进程最多只能是它的儿子进程。而这个子进程是怎么创建的，`fork` 调用，而不是之前提到的 `execve` 调用。前者创建一个子进程，后者则会覆盖当前进程。因为我们发现 `/sbin/getty` 运行时，`init` 并没有退出，因此可以判断是 `fork` 调用创建一个子进程后，才通过 `execve` 执行了 `/sbin/getty`。

因此，可以总结出这么一个调用过程：

```
     fork     execve         execve         fork           execv
e
init --> init --> /sbin/getty --> /bin/login --> /bin/login -->
/bin/bash
```

这里的 `execve` 调用以后，后者将直接替换前者，因此当键入 `exit` 退出 `/bin/bash` 以后，也就相当于 `/sbin/getty` 都已经结束了，因此最前面的 `init` 程序判断 `/sbin/getty` 退出了，又会创建一个子进程把 `/sbin/getty` 启动，进而又启动了 `/bin/login`，又看到了那个 "XXX login:"。

通过 `ps` 和 `pstree` 命令看看实际情况是不是这样，前者打印出进程的信息，后者则打印出调用关系。

```
$ ps -ef | egrep "/sbin/init|/sbin/getty|bash|/bin/login"
root         1     0  0 21:43 ?        00:00:01 /sbin/init
root      3957     1  0 21:43 tty4     00:00:00 /sbin/getty 3840
0 tty4
root      3958     1  0 21:43 tty5     00:00:00 /sbin/getty 3840
0 tty5
root      3963     1  0 21:43 tty3     00:00:00 /sbin/getty 3840
0 tty3
root      3965     1  0 21:43 tty6     00:00:00 /sbin/getty 3840
0 tty6
root      7023     1  0 22:48 tty1     00:00:00 /sbin/getty 3840
0 tty1
root      7081     1  0 22:51 tty2     00:00:00 /bin/login --
falcon    7092  7081  0 22:52 tty2     00:00:00 -bash
```

上面的结果已经过滤了一些不相干的数据。从上面的结果可以看到，除了 `tty2` 被替换成 `/bin/login` 外，其他终端都运行着 `/sbin/getty`，说明终端 2 上的进程是 `/bin/login`，它已经把 `/sbin/getty` 替换掉，另外，我们看到 `-bash` 进程的父进程是 `7081` 刚好是 `/bin/login` 程序，这说明 `/bin/login` 启动了 `-bash`，但是它并没有替换掉 `/bin/login`，而是成为了 `/bin/login` 的子进程，这说明 `/bin/login` 通过 `fork` 创建了一个子进程并通过 `execve`

执行了 `-bash` （后者通过 `strace` 跟踪到）。而 `init` 呢，其进程 ID 是 1，是 `/sbin/getty` 和 `/bin/login` 的父进程，说明 `init` 启动或者间接启动了它们。下面通过 `pstree` 来查看调用树，可以更清晰地看出上述关系。

```
$ pstree | egrep "init|getty|\-bash|login"
init-+-5*[getty]
     |-login---bash
     |-xfce4-terminal-+-bash-+-grep
```

结果显示 `init` 是 5 个 `getty` 程序，`login` 程序和 `xfce4-terminal` 的父进程，而后两者则是 `bash` 的父进程，另外我们执行的 `grep` 命令则在 `bash` 上运行，是 `bash` 的子进程，这个将是我们后面关心的问题。

从上面的结果发现，`init` 作为所有进程的父进程，它的父进程 ID 饶有兴趣的是 0，它是怎么被启动的呢？谁才是真正的“造物主”？

谁启动了 /sbin/init

如果用过 `Lilo` 或者 `Grub` 这些操作系统引导程序，可能会用到 Linux 内核的一个启动参数 `init` ，当忘记密码时，可能会把这个参数设置成 `/bin/bash` ，让系统直接进入命令行，而无须输入帐号和密码，这样就可以方便地把登录密码修改掉。

这个 `init` 参数是个什么东西呢？通过 `man bootparam` 会发现它的秘密， `init` 参数正好指定了内核启动后要启动的第一个程序，而如果没有指定该参数，内核将依次查找 `/sbin/init` ， `/etc/init` ， `/bin/init` ， `/bin/sh` ，如果找不到这几个文件中的任何一个，内核就要恐慌（panic）了，并挂（hang）在那里一动不动了（注：如果 `panic=timeout` 被传递给内核并且 `timeout` 大于 0，那么就不会挂住而是重启）。

因此 `/sbin/init` 就是 Linux 内核启动的。而 Linux 内核呢？是通过 `Lilo` 或者 `Grub` 等引导程序启动的， `Lilo` 和 `Grub` 都有相应的配置文件，一般对应 `/etc/lilo.conf` 和 `/boot/grub/menu.lst` ，通过这些配置文件可以指定内核映像文件、系统根目录所在分区、启动选项标签等信息，从而能够让它们顺利把内核启动起来。

那 `Lilo` 和 `Grub` 本身又是怎么被运行起来的呢？有了解 MBR 不？MBR 就是主引导扇区，一般情况下这里存放着 `Lilo` 和 `Grub` 的代码，而谁知道正好是这里存放了它们呢？BIOS，如果你用光盘安装过操作系统的话，那么应该修改过 `BIOS` 的默认启动设置，通过设置可以让系统从光盘、硬盘、U 盘甚至软盘启动。正是这里的设置让 BIOS 知道了 MBR 处的代码需要被执行。

那 BIOS 又是什么时候被起来的呢？处理器加电后有一个默认的起始地址，一上电就执行到了这里，再之前就是开机键按键后的上电时序。

更多系统启动的细节，看看 `man boot-scripts` 吧。

到这里， `/bin/bash` 的神秘面纱就被揭开了，它只是系统启动后运行的一个程序而已，只不过这个程序可以响应用户的请求，那它到底是如何响应用户请求的呢？

/bin/bash 如何处理用户键入的命令

预备知识

在执行磁盘上某个程序时，通常不会指定这个程序文件的绝对路径，比如要执行 `echo` 命令时，一般不会输入 `/bin/echo` ，而仅仅是输入 `echo` 。那为什么这样 `bash` 也能够找到 `/bin/echo` 呢？原因是 Linux 操作系统支持这样一种策略：Shell 的一个环境变量 `PATH` 里头存放了程序的一些路径，当 Shell 执行程序时有可能去这些目录下查找。 `which` 作为 Shell（这里特指 `bash` ）的一个内置命令，如果用户输入的命令是磁盘上的某个程序，它会返回这个文件的全路径。

有三个东西和终端的关系很大，那就是标准输入、标准输出和标准错误，它们是三个文件描述符，一般对应描述符 0，1，2。在 C 语言程序里，我们可以把它们当作文件描述符一样进行操作。在命令行下，则可以使用重定向字符 `>，<` 等对它们进行操作。对于标准输出和标准错误，都默认输出到终端，对于标准输入，也同样默认从终端输入。

哪种命令先被执行

在 C 语言里头要写一段输入字符串的命令很简单，调用 `scanf` 或者 `fgets` 就可以。这个在 `bash` 里头应该是类似的。但是它获取用户的命令以后，如何分析命令，如何响应不同的命令呢？

首先来看看 `bash` 下所谓的命令，用最常见的 `test` 来作测试。

- 字符串被解析成命令

  随便键入一个字符串 `test1`， `bash` 发出响应，告知找不到这个程序：

```
$ test1
bash: test1: command not found
```

- 内置命令

  而当键入 `test` 时，看不到任何输出，唯一响应是，新命令提示符被打印了：

```
$ test
$
```

  查看 `test` 这个命令的类型，即查看 `test` 将被如何解释， `type` 告诉我们 `test` 是一个内置命令，如果没有理解错， `test` 应该是利用诸如 `case "test": do something;break;` 这样的机制实现的，具体如何实现可以查看 `bash` 源代码。

```
$ type test
test is a shell builtin
```

- 外部命令

  这里通过 `which` 查到 `/usr/bin` 下有一个 `test` 命令文件，在键入 `test` 时，到底哪一个被执行了呢？

```
$ which test
/usr/bin/test
```

  执行这个呢？也没什么反应，到底谁先被执行了？

```
$ /usr/bin/test
```

从上述演示中发现一个问题？如果输入一个命令，这个命令要么就不存在，要么可能同时是 Shell 的内置命令、也有可能是磁盘上环境变量 `PATH` 所指定的目录下的某个程序文件。

考虑到 `test` 内置命令和 `/usr/bin/test` 命令的响应结果一样，我们无法知道哪一个先被执行了，怎么办呢？把 `/usr/bin/test` 替换成一个我们自己的命令，并让它打印一些信息(比如 `hello, world!`)，这样我们就知道到底谁被执行了。写完程序，编译好，命名为 `test` 放到 `/usr/bin` 下（记得备份原来那个）。开始测试：

键入 `test` ，还是没有效果：

```
$ test
$
```

而键入绝对路径呢，则打印了 `hello, world!` 诶，那默认情况下肯定是内置命令先被执行了：

```
$ /usr/bin/test
hello, world!
```

由上述实验结果可见，内置命令比磁盘文件中的程序优先被 `bash` 执行。原因应该是内置命令避免了不必要的 `fork/execve` 调用，对于采用类似算法实现的功能，内置命令理论上有更高运行效率。

下面看看更多有趣的内容，键盘键入的命令还有可能是什么呢？因为 `bash` 支持别名（ `alias` ）和函数（ `function` ），所以还有可能是别名和函数，另外，如果 `PATH` 环境变量指定的不同目录下有相同名字的程序文件，那到底哪个被优先找到呢？

下面再作一些实验，

- 别名

  把 `test` 命名为 `ls -l` 的别名，再执行 `test` ，竟然执行了 `ls -l` ，说明别名（ `alias` ）比内置命令（ `builtin` ）更优先：

```
$ alias test="ls -l"
$ test
total 9488
drwxr-xr-x 12 falcon falcon    4096 2008-02-21 23:43 bash-
3.2
-rw-r--r--  1 falcon falcon 2529838 2008-02-21 23:30 bash-
3.2.tar.gz
```

- 函数

定义一个名叫 `test` 的函数，执行一下，发现，还是执行了 `ls -l` ，说明 `function` 没有 `alias` 优先级高：

```
$ function test { echo "hi, I'm a function"; }
$ test
total 9488
drwxr-xr-x 12 falcon falcon    4096 2008-02-21 23:43 bash-
3.2
-rw-r--r--  1 falcon falcon 2529838 2008-02-21 23:30 bash-
3.2.tar.gz
```

把别名给去掉（ `unalias` ），现在执行的是函数，说明函数的优先级比内置命令也要高：

```
$ unalias test
$ test
hi, I'm a function
```

如果在命令之前跟上 `builtin` ，那么将直接执行内置命令：

```
$ builtin test
```

要去掉某个函数的定义，这样就可以：

```
$ unset test
```

通过这个实验我们得到一个命令的别名（`alias`）、函数（`function`），内置命令（`builtin`）和程序（`program`）的执行优先次序：

```
    先    alias --> function --> builtin --> program   后
```

实际上，`type` 命令会告诉我们这些细节，`type -a` 会按照 `bash` 解析的顺序依次打印该命令的类型，而 `type -t` 则会给出第一个将被解析的命令的类型，之所以要做上面的实验，是为了让大家加印象。

```
$ type -a test
test is a shell builtin
test is /usr/bin/test
$ alias test="ls -l"
$ function test { echo "I'm a function"; }
$ type -a test
test is aliased to `ls -l'
test is a function
test ()
{
    echo "I'm a function"
}
test is a shell builtin
test is /usr/bin/test
$ type -t test
alias
```

下面再看看 `PATH` 指定的多个目录下有同名程序的情况。再写一个程序，打印 `hi, world!`，以示和 `hello, world!` 的区别，放到 `PATH` 指定的另外一个目录 `/bin` 下，为了保证测试的说服力，再写一个放到另外一个叫 `/usr/local/sbin` 的目录下。

先看看 `PATH` 环境变量，确保它有 `/usr/bin`，`/bin` 和 `/usr/local/sbin` 这几个目录，然后通过 `type -P`（`-P` 参数强制到 `PATH` 下查找，而不管是别名还是内置命令等，可以通过 `help type` 查看该参数的含义）查看，到底哪个先被执行。

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/us
r/games
$ type -P test
/usr/local/sbin/test
```

如上可以看到 `/usr/local/sbin` 下的先被找到。

把 `/usr/local/sbin/test` 下的给删除掉，现在 `/usr/bin` 下的先被找到：

```
$ rm /usr/local/sbin/test
$ type -P test
/usr/bin/test
```

`type -a` 也显示类似的结果：

```
$ type -a test
test is aliased to `ls -l'
test is a function
test ()
{
    echo "I'm a function"
}
test is a shell builtin
test is /usr/bin/test
test is /bin/test
```

因此，可以找出这么一个规律：Shell 从 `PATH` 列出的路径中依次查找用户输入的命令。考虑到程序的优先级最低，如果想优先执行磁盘上的程序文件 `test` 呢？那么就可以用 `test -P` 找出这个文件并执行就可以了。

补充：对于 Shell 的内置命令，可以通过 `help command` 的方式获得帮助，对于程序文件，可以查看用户手册（当然，这个需要安装，一般叫做 `xxx-doc`），`man command`。

这些特殊字符是如何解析的：`|, >, <, &`

在命令行上，除了输入各种命令以及一些参数外，比如上面 `type` 命令的各种参数 `-a` ， `-P` 等，对于这些参数，是传递给程序本身的，非常好处理，比如 `if` ， `else` 条件分支或者 `switch` ， `case` 都可以处理。当然，在 `bash` 里头可能使用专门的参数处理函数 `getopt` 和 `getopt_long` 来处理它们。

而 `|` ， `>` ， `<` ， `&` 等字符，则比较特别， Shell 是怎么处理它们的呢？它们也被传递给程序本身吗？可我们的程序内部一般都不处理这些字符的，所以应该是 Shell 程序自己解析了它们。

先来看看这几个字符在命令行的常见用法，

`<` 字符表示：把 `test.c` 文件重定向为标准输入，作为 `cat` 命令输入，而 `cat` 默认输出到标准输出：

```
$ cat < ./test.c
#include <stdio.h>

int main(void)
{
        printf("hi, myself!\n");
        return 0;
}
```

`>` 表示把标准输出重定向为文件 `test_new.c` ，结果内容输出到 `test_new.c` ：

```
$ cat < ./test.c > test_new.c
```

对于 `>` ， `<` ， `>>` ， `<<` ， `<>` 我们都称之为重定向（ `redirect` ）， Shell 到底是怎么进行所谓的“重定向”的呢？

这主要归功于 `dup/fcntl` 等函数，它们可以实现：复制文件描述符，让多个文件描述符共享同一个文件表项。比如，当把文件 `test.c` 重定向为标准输入时。假设之前用以打开 `test.c` 的文件描述符是 5 ，现在就把 5 复制为了 0 ，这样当 `cat` 试图从标准输入读出内容时，也就访问了文件描述符 5 指向的文件表项，接着读出了文件内容。输出重定向与此类似。其他的重定向，诸如 `>>` ， `<<` ， `<>` 等虽然和 `>` ， `<` 的具体实现功能不太一样，但本质是一样的，都是文件

描述符的复制，只不过可能对文件操作有一些附加的限制，比如 `>>` 在输出时追加到文件末尾，而 `>` 则会从头开始写入文件，前者意味着文件的大小会增长，而后者则意味文件被重写。

那么 `|` 呢？ `|` 被形象地称为“管道”，实际上它就是通过 C 语言里头的无名管道来实现的。先看一个例子，

```
$ cat < ./test.c  | grep hi
        printf("hi, myself!\n");
```

在这个例子中， `cat` 读出了 `test.c` 文件中的内容，并输出到标准输出上，但是实际上输出的内容却只有一行，原因是这个标准输出被“接到”了 `grep` 命令的标准输入上，而 `grep` 命令只打印了包含 “hi” 字符串的一行。

这是怎么被“接”上的。 `cat` 和 `grep` 作为两个单独的命令，它们本身没有办法把两者的输入和输出“接”起来。这正是 Shell 自己的“杰作”，它通过 C 语言里头的 `pipe` 函数创建了一个管道（一个包含两个文件描述符的整形数组，一个描述符用于写入数据，一个描述符用于读入数据），并且通过 `dup/fcntl` 把 `cat` 的输出复制到了管道的输入，而把管道的输出则复制到了 `grep` 的输入。这真是一个奇妙的想法。

那 `&` 呢？当你在程序的最后跟上这个奇妙的字符以后就可以接着做其他事情了，看看效果：

```
$ sleep 50 & #让程序在后台运行
[1] 8261
```

提示符被打印出来，可以输入东西，让程序到前台运行，无法输入东西了，按下 `CTRL+Z` ，再让程序到后台运行：

```
$ fg %1
sleep 50

[1]+  Stopped                 sleep 50
```

实际上 `&` 正是 `Shell` 支持作业控制的表征，通过作业控制，用户在命令行上可以同时作几个事情（把当前不做的放到后台，用 `&` 或者 `CTRL+Z` 或者 `bg`）并且可以自由地选择当前需要执行哪一个（用 `fg` 调到前台）。这在实现时应该涉及到很多东西，包括终端会话（ `session` ）、终端信号、前台进程、后台进程等。而在命令的后面加上 `&` 后，该命令将被作为后台进程执行，后台进程是什么呢？这类进程无法接收用户发送给终端的信号（如 `SIGHUP` ， `SIGQUIT` ， `SIGINT` ），无法响应键盘输入（被前台进程占用着），不过可以通过 `fg` 切换到前台而享受作为前台进程具有的特权。

因此，当一个命令被加上 `&` 执行后，Shell 必须让它具有后台进程的特征，让它无法响应键盘的输入，无法响应终端的信号（意味忽略这些信号），并且比较重要的是新的命令提示符得打印出来，并且让命令行接口可以继续执行其他命令，这些就是 Shell 对 `&` 的执行动作。

还有什么神秘的呢？你也可以写自己的 Shell 了，并且可以让内核启动后就执行它 `l` ，在 `lilo` 或者 `grub` 的启动参数上设置 `init=/path/to/your/own/shell/program` 就可以。当然，也可以把它作为自己的登录 Shell ，只需要放到 `/etc/passwd` 文件中相应用户名所在行的最后就可以。不过貌似到现在还没介绍 Shell 是怎么执行程序，是怎样让程序变成进程的，所以继续。

/bin/bash 用什么魔法让一个普通程序变成了进程

当我们从键盘键入一串命令，Shell 奇妙地响应了，对于内置命令和函数，Shell 自身就可以解析了（通过 `switch` ， `case` 之类的 C 语言语句）。但是，如果这个命令是磁盘上的一个文件呢。它找到该文件以后，怎么执行它的呢？

还是用 `strace` 来跟踪一个命令的执行过程看看。

```
$ strace -f -o strace.log /usr/bin/test
hello, world!
$ cat strace.log | sed -ne "1p"   #我们对第一行很感兴趣
8445  execve("/usr/bin/test", ["/usr/bin/test"], [/* 33 vars */]
) = 0
```

从跟踪到的结果的第一行可以看到 `bash` 通过 `execve` 调用了 `/usr/bin/test` ，并且给它传了 33 个参数。这 33 个 `vars` 是什么呢？看看 `declare -x` 的结果（这个结果只有 32 个，原因是 `vars` 的最后一个变量需要是一个结束标志，即 `NULL` ）。

```
$ declare -x | wc -l   #declare -x声明的环境变量将被导出到子进程中
32
$ export TEST="just a test"   #为了认证declare -x和之前的vars的个数的关系，再加一个
$ declare -x | wc -l
33
$ strace -f -o strace.log /usr/bin/test   #再次跟踪，看看这个关系
hello, world!
$ cat strace.log | sed -ne "1p"
8523  execve("/usr/bin/test", ["/usr/bin/test"], [/* 34 vars */]
) = 0
```

通过这个演示发现，当前 Shell 的环境变量中被设置为 `export` 的变量被复制到了新的程序里头。不过虽然我们认为 Shell 执行新程序时是在一个新的进程里头执行的，但是 `strace` 并没有跟踪到诸如 `fork` 的系统调用（可能是 `strace` 自己设计的时候并没有跟踪 `fork` ，或者是在 `fork` 之后才跟踪）。但是有一个事实我们不得不承认：当前 Shell 并没有被新程序的进程替换，所以说 Shell 肯定是先调用 `fork` （也有可能是 `vfork` ）创建了一个子进程，然后再调用 `execve` 执行新程序的。如果你还不相信，那么直接通过 `exec` 执行新程序看看，这个可是直接把当前 Shell 的进程替换掉的。

```
exec /usr/bin/test
```

该可以看到当前 Shell “哗”（听不到，突然没了而已）的一下就没有了。

下面来模拟一下 Shell 执行普通程序。 `multiprocess` 相当于当前 Shell ，而 `/usr/bin/test` 则相当于通过命令行传递给 Shell 的一个程序。这里是代码：

```
/* multiprocess.c */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>     /* sleep, fork, _exit */

int main()
{
    int child;
    int status;

    if( (child = fork()) == 0) {    /* child */
        printf("child: my pid is %d\n", getpid());
        printf("child: my parent's pid is %d\n", getppid());
        execlp("/usr/bin/test","/usr/bin/test",(char *)NULL);;
    } else if(child < 0){       /* error */
            printf("create child process error!\n");
            _exit(0);
    }                                                   /* paren
t */
    printf("parent: my pid is %d\n", getpid());
    if ( wait(&status) == child ) {
        printf("parent: wait for my child exit successfully!\n")
;
    }
}
```

运行看看，

```
$ make multiprocess
$ ./multiprocess
child: my pid is 2251
child: my parent's pid is 2250
hello, world!
parent: my pid is 2250
parent: wait for my child exit successfully!
```

从执行结果可以看出，`/usr/bin/test` 在 `multiprocess` 的子进程中运行并不干扰父进程，因为父进程一直等到了 `/usr/bin/test` 执行完成。

再回头看看代码，你会发现 `execlp` 并没有传递任何环境变量信息给 `/usr/bin/test` ，到底是怎么把环境变量传送过去的呢？通过 `man exec` 我们可以看到一组 `exec` 的调用，在里头并没有发现 `execve` ，但是通过 `man execve` 可以看到该系统调用。实际上 `exec` 的那一组调用都只是 `libc` 库提供的，而 `execve` 才是真正的系统调用，也就是说无论使用 `exec` 调用中的哪一个，最终调用的都是 `execve` ，如果使用 `execlp` ，那么 `execlp` 将通过一定的处理把参数转换为 `execve` 的参数。因此，虽然我们没有传递任何环境变量给 `execlp` ，但是默认情况下， `execlp` 把父进程的环境变量复制给了子进程，而这个动作是在 `execlp` 函数内部完成的。

现在，总结一下 `execve` ，它有有三个参数，

`-` 第一个是程序本身的绝对路径，对于刚才使用的 `execlp` ，我们没有指定路径，这意味着它会设法到 `PATH` 环境变量指定的路径下去寻找程序的全路径。 `-` 第二个参数是一个将传递给被它执行的程序的参数数组指针。正是这个参数把我们从命令行上输入的那些参数，诸如 `grep` 命令的 `-v` 等传递给了新程序，可以通过 `main` 函数的第二个参数 `char` *`argv[]`* 获得这些内容。 `-` 第三个参数是一个将传递给被它执行的程序的环境变量，这些环境变量也可以通过 *`main`* 函数的第三个变量获取，只要定义一个 *`char`* `env[]` 就可以了，只是通常不直接用它罢了，而是通过另外的方式，通过 `extern char` ** `environ` 全局变量（环境变量表的指针）或者 `getenv` 函数来获取某个环境变量的值。

当然，实际上，当程序被 `execve` 执行后，它被加载到了内存里，包括程序的各种指令、数据以及传递给它的各种参数、环境变量等都被存放在系统分配给该程序的内存空间中。

我们可以通过 `/proc/<pid>/maps` 把一个程序对应的进程的内存映象看个大概。

```
$ cat /proc/self/maps   #查看cat程序自身加载后对应进程的内存映像
08048000-0804c000 r-xp 00000000 03:01 273716     /bin/cat
0804c000-0804d000 rw-p 00003000 03:01 273716     /bin/cat
0804d000-0806e000 rw-p 0804d000 00:00 0          [heap]
b7c46000-b7e46000 r--p 00000000 03:01 87528      /usr/lib/locale
/locale-archive
b7e46000-b7e47000 rw-p b7e46000 00:00 0
b7e47000-b7f83000 r-xp 00000000 03:01 466875     /lib/libc-2.5.s
o
b7f83000-b7f84000 r--p 0013c000 03:01 466875     /lib/libc-2.5.s
o
b7f84000-b7f86000 rw-p 0013d000 03:01 466875     /lib/libc-2.5.s
o
b7f86000-b7f8a000 rw-p b7f86000 00:00 0
b7fa1000-b7fbc000 r-xp 00000000 03:01 402817     /lib/ld-2.5.so
b7fbc000-b7fbe000 rw-p 0001b000 03:01 402817     /lib/ld-2.5.so
bfcdf000-bfcf4000 rw-p bfcdf000 00:00 0          [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]
```

关于程序加载和进程内存映像的更多细节请参考《C 语言程序缓冲区注入分析》。

到这里，关于命令行的秘密都被“曝光”了，可以开始写自己的命令行解释程序了。

关于进程的相关操作请参考《进程与进程的基本操作》。

补充：上面没有讨论到一个比较重要的内容，那就是即使 `execve` 找到了某个可执行文件，如果该文件属主没有运行该程序的权限，那么也没有办法运行程序。可通过 `ls -l` 查看程序的权限，通过 `chmod` 添加或者去掉可执行权限。

文件属主具有可执行权限时才可以执行某个程序：

```
$ whoami
falcon
$ ls -l hello  #查看用户权限(第一个x表示属主对该程序具有可执行权限
-rwxr-xr-x 1 falcon users 6383 2000-01-23 07:59 hello*
$ ./hello
Hello World
$ chmod -x hello  #去掉属主的可执行权限
$ ls -l hello
-rw-r--r-- 1 falcon users 6383 2000-01-23 07:59 hello
$ ./hello
-bash: ./hello: Permission denied
```

参考资料

- Linux 启动过程：`man boot-scripts`
- Linux 内核启动参数：`man bootparam`
- `man 5 passwd`
- `man shadow`
- 《UNIX 环境高级编程》，进程关系一章

动态符号链接的细节

前言

Linux 支持动态链接库，不仅节省了磁盘、内存空间，而且可以提高程序运行效率。不过引入动态链接库也可能会带来很多问题，例如动态链接库的调试、升级更新和潜在的安全威胁[1], [2]。这里主要讨论符号的动态链接过程，即程序在执行过程中，对其中包含的一些未确定地址的符号进行重定位的过程[1], [2]。

本篇主要参考资料[3]和[8]，前者侧重实践，后者侧重原理，把两者结合起来就方便理解程序的动态链接过程了。另外，动态链接库的创建、使用以及调用动态链接库的部分参考了资料[1], [2]。

下面先来看看几个基本概念，接着就介绍动态链接库的创建、隐式和显示调用，最后介绍符号的动态链接细节。

基本概念

ELF

`ELF` 是 Linux 支持的一种程序文件格式，本身包含重定位、执行、共享（动态链接库）三种类型（`man elf`）。

代码：

```
/* test.c */
#include <stdio.h>

int global = 0;

int main()
{
        char local = 'A';

        printf("local = %c, global = %d\n", local, global);

        return 0;
}
```

演示：

通过 `-c` 生成可重定位文件 `test.o`，这里不会进行链接：

```
$ gcc -c test.c
$ file test.o
test.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV
), not stripped
```

链接后才可以执行：

```
$ gcc -o test test.o
$ file test
test: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), not stripped
```

也可链接成动态链接库，不过一般不会把 `main` 函数链接成动态链接库，后面再介绍：

```
$ gcc -fpic -shared -Wl,-soname,libtest.so.0 -o libtest.so.0.0 t
est.o
$ file libtest.so.0.0
libtest.so.0.0: ELF 32-bit LSB shared object, Intel 80386, versi
on 1 (SYSV), not stripped
```

虽然 `ELF` 文件本身就支持三种不同的类型，不过它有一个统一的结构。这个结构是：

```
文件头部(ELF Header)
程序头部表(Program Header Table)
节区1(Section1)
节区2(Section2)
节区3(Section3)
...
节区头部表(Section Header Table)
```

无论是文件头部、程序头部表、节区头部表，还是节区，它们都对应着 C 语言里头的一些结构体（ `elf.h` 中定义）。文件头部主要描述 `ELF` 文件的类型，大小，运行平台，以及和程序头部表和节区头部表相关的信息。节区头部表则用于可重定位文件，以便描述各个节区的信息，这些信息包括节区的名字、类型、大小等。程序头部表则用于描述可执行文件或者动态链接库，以便系统加载和执行它们。而节区主要存放各种特定类型的信息，比如程序的正文区（代码）、数据区（初始化和未初始化的数据）、调试信息、以及用于动态链接的一些节区，比如解释器（ `.interp` ）节区将指定程序动态装载 `/` 链接器 `ld-linux.so` 的位置，而过程链接表（ `plt` ）、全局偏移表（ `got` ）、重定位表则用于辅助动态链接过程。

符号

对于可执行文件除了编译器引入的一些符号外，主要就是用户自定义的全局变量，函数等，而对于可重定位文件仅仅包含用户自定义的一些符号。

- 生成可重定位文件

```
$ gcc -c test.c
$ nm test.o
00000000 B global
00000000 T main
         U printf
```

  上面包含全局变量、自定义函数以及动态链接库中的函数，但不包含局部变量，而且发现这三个符号的地址都没有确定。

  注： `nm` 命令可用来查看 `ELF` 文件的符号表信息。

- 生成可执行文件

```
$ gcc -o test test.o
$ nm test | egrep "main$| printf|global$"
080495a0 B global
08048354 T main
         U printf@@GLIBC_2.0

经链接，`global` 和 `main` 的地址都已经确定了，但是 `printf` 却还没，因为它是动态链接库 `glibc` 中定义函数，需要动态链接，而不是这里的“静态”链接。
```

重定位：是将符号引用与符号定义进行链接的过程

从上面的演示可以看出，重定位文件 `test.o` 中的符号地址都是没有确定的，而经过静态链接（ `gcc` 默认调用 `ld` 进行链接）以后有两个符号地址已经确定了，这样一个确定符号地址的过程实际上就是链接的实质。链接过后，对符号的引用变成了对地址（定义符号时确定该地址）的引用，这样程序运行时就可通过访问内存地址而访问特定的数据。

我们也注意到符号 `printf` 在可重定位文件和可执行文件中的地址都没有确定，这意味着该符号是一个外部符号，可能定义在动态链接库中，在程序运行时需要通过动态链接器（ `ld-linux.so` ）进行重定位，即动态链接。

通过这个演示可以看出 `printf` 确实在 `glibc` 中有定义。

```
$ nm -D /lib/`uname -m`-linux-gnu/libc.so.6 | grep "\ printf$"
0000000000053840 T printf
```

除了 `nm` 以外，还可以用 `readelf -s` 查看 `.dynsym` 表或者用 `objdump -tT` 查看。

需要提到的是，用 `nm` 命令不带 `-D` 参数的话，在较新的系统上已经没有办法查看 `libc.so` 的符号表了，因为 `nm` 默认打印常规符号表（在 `.symtab` 和 `.strtab` 节区中），但是，在打包时为了减少系统大小，这些符号已经被 `strip` 掉了，只保留了动态符号（在 `.dynsym` 和 `.dynstr` 中）以便动态链接器在执行程序时寻址这些外部用到的符号。而常规符号除了动态符号以外，还包含有一些静态符号，比如说本地函数，这个信息主要是调试器会用，对于正常部署的系统，一般会用 `strip` 工具删除掉。

关于 `nm` 与 `readelf -s` 的详细比较，可参考：nm vs “readelf -s”。

动态链接

动态链接就是在程序运行时对符号进行重定位，确定符号对应的内存地址的过程。

Linux 下符号的动态链接默认采用Lazy Mode方式，也就是说在程序运行过程中用到该符号时才去解析它的地址。这样一种符号解析方式有一个好处：只解析那些用到的符号，而对那些不用的符号则永远不用解析，从而提高程序的执行效率。

不过这种默认是可以通过设置 `LD_BIND_NOW` 为非空来打破的（下面会通过实例来分析这个变量的作用），也就是说如果设置了这个变量，动态链接器将在程序加载后和符号被使用之前就对这些符号的地址进行解析。

动态链接库

上面提到重定位的过程就是对符号引用和符号地址进行链接的过程，而动态链接过程涉及到的符号引用和符号定义分别对应可执行文件和动态链接库，在可执行文件中可能引用了某些动态链接库中定义的符号，这类符号通常是函数。

为了让动态链接器能够进行符号的重定位，必须把动态链接库的相关信息写入到可执行文件当中，这些信息是什么呢？

```
$ readelf -d test | grep NEEDED
 0x00000001 (NEEDED)                     Shared library: [libc.s
o.6]
```

`ELF` 文件有一个特别的节区： `.dynamic` ，它存放了和动态链接相关的很多信息，例如动态链接器通过它找到该文件使用的动态链接库。不过，该信息并未包含动态链接库 `libc.so.6` 的绝对路径，那动态链接器去哪里查找相应的库呢？

通过 `LD_LIBRARY_PATH` 参数，它类似 Shell 解释器中用于查找可执行文件的 `PATH` 环境变量，也是通过冒号分开指定了各个存放库函数的路径。该变量实际上也可以通过 `/etc/ld.so.conf` 文件来指定，一行对应一个路径名。为了提高查找和加载动态链接库的效率，系统启动后会通过 `ldconfig` 工具创建一个库的缓存 `/etc/ld.so.cache` 。如果用户通过 `/etc/ld.so.conf` 加入了新的库搜索路径或者是把新库加到某个原有的库目录下，最好是执行一下 `ldconfig` 以便刷新缓存。

需要补充的是，因为动态链接库本身还可能引用其他的库，那么一个可执行文件的动态符号链接过程可能涉及到多个库，通过 `readelf -d` 可以打印出该文件直接依赖的库，而通过 `ldd` 命令则可以打印出所有依赖或者间接依赖的库。

```
$ ldd test
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/libc.so.6 (0xb7da2000)
        /lib/ld-linux.so.2 (0xb7efc000)
```

`libc.so.6` 通过 `readelf -d` 就可以看到的，是直接依赖的库；而 `linux-gate.so.1` 在文件系统中并没有对应的库文件，它是一个虚拟的动态链接库，对应进程内存映像的内核部分，更多细节请参考资料[11]; 而 `/lib/ld-linux.so.2` 正好是动态链接器，系统需要用它来进行符号重定位。那 `ldd` 是怎么知道 `/lib/ld-linux.so` 就是该文件的动态链接器呢？

那是因为 `ELF` 文件通过专门的节区指定了动态链接器，这个节区就是 `.interp` 。

```
$ readelf -x .interp test

Hex dump of section '.interp':
  0x08048114 2f6c6962 2f6c642d 6c696e75 782e736f /lib/ld-linux.s
o
  0x08048124 2e3200                              .2.
```

可以看到这个节区刚好有字符串 `/lib/ld-linux.so.2` ，即 `ld-linux.so` 的绝对路径。

我们发现，与 `libc.so` 不同的是， `ld-linux.so` 的路径是绝对路径，而 `libc.so` 仅仅包含了文件名。原因是：程序被执行时， `ld-linux.so` 将最先被装载到内存中，没有其他程序知道去哪里查找 `ld-linux.so` ，所以它的路径必须是绝对的；当 `ld-linux.so` 被装载以后，由它来去装载可执行文件和相关的共享库，它将根据 `PATH` 变量和 `LD_LIBRARY_PATH` 变量去磁盘上查找它们，因此可执行文件和共享库都可以不指定绝对路径。

下面着重介绍动态链接器本身。

动态链接器（dynamic linker/loader）

Linux 下 `elf` 文件的动态链接器是 `ld-linux.so` ，即 `/lib/ld-linux.so.2` 。从名字来看和静态链接器 `ld` （ `gcc` 默认使用的链接器，见参考资料[10]）类似。通过 `man ld-linux` 可以获取与动态链接器相关的资料，包括各种相关的环境变量和文件都有详细的说明。

对于环境变量，除了上面提到过的 `LD_LIBRARY_PATH` 和 `LD_BIND_NOW` 变量外，还有其他几个重要参数，比如 `LD_PRELOAD` 用于指定预装载一些库，以便替换其他库中的函数，从而做一些安全方面的处理 [6]，[9]，[12]，而环境变量 `LD_DEBUG` 可以用来进行动态链接的相关调试。

对于文件，除了上面提到的 `ld.so.conf` 和 `ld.so.cache` 外，还有一个文件 `/etc/ld.so.preload` 用于指定需要预装载的库。

从上一小节中发现有一个专门的节区 `.interp` 存放有动态链接器，但是这个节区为什么叫做 `.interp` （ `interpeter` ） 呢？因为当 `Shell` 解释器或者其他父进程通过 exec 启动我们的程序时，系统会先为 ld-linux 创建内存映像，然后把控制权交给 ld-linux ，之后 ld-linux 负责为可执行程序提供运行环境，负责解释程序的运行，因此 ld-linux 也叫做 dynamic loader （或 intepreter`）（关于程序的加载过程请参考资料 [13]）

那么在 `exec` （）之后和程序指令运行之前的过程是怎样的呢？ `ld-linux.so` 主要为程序本身创建了内存映像（以下内容摘自资料 [8]），大体过程如下：

- 将可执行文件的内存段添加到进程映像中；
- 把共享目标内存段添加到进程映像中；
- 为可执行文件和它的共享目标（动态链接库）执行重定位操作；
- 关闭用来读入可执行文件的文件描述符，如果动态链接程序收到过这样的文件描述符的话；
- 将控制转交给程序，使得程序好像从 `exec()` 直接得到控制

关于第 1 步，在 `ELF` 文件的文件头中就指定了该文件的入口地址，程序的代码和数据部分会相继 `map` 到对应的内存中。而关于可执行文件本身的路径，如果指定了 `PATH` 环境变量， `ld-linux` 会到 `PATH` 指定的相关目录下查找。

```
$ readelf -h test | grep Entry
  Entry point address:               0x80482b0
```

对于第 2 步，上一节提到的 `.dynamic` 节区指定了可执行文件依赖的库名， `ld-linux` （在这里叫做动态装载器或程序解释器比较合适）再从 `LD_LIBRARY_PATH` 指定的路径中找到相关的库文件或者直接从 `/etc/ld.so.cache` 库缓冲中加载相关库到内存中。（关于进程的内存映像，推荐参考资料 [14]）

对于第 3 步，在前面已提到，如果设置了 `LD_BIND_NOW` 环境变量，这个动作就会在此时发生，否则将会采用 `lazy mode` 方式，即当某个符号被使用时才会进行符号的重定位。不过无论在什么时候发生这个动作，重定位的过程大体是一样的（在后面将主要介绍该过程）。

对于第 4 步，这个主要是释放文件描述符。

对于第 5 步，动态链接器把程序控制权交还给程序。

现在关心的主要是第 3 步，即如何进行符号的重定位？下面来探求这个过程。期间会逐步讨论到和动态链接密切相关的三个数据结构，它们分别是 `ELF` 文件的过程链接表、全局偏移表和重定位表，这三个表都是 `ELF` 文件的节区。

过程链接表（plt）

从上面的演示发现，还有一个 `printf` 符号的地址没有确定，它应该在动态链接库 `libc.so` 中定义，需要进行动态链接。这里假设采用 `lazy mode` 方式，即执行到 `printf` 所在位置时才去解析该符号的地址。

假设当前已经执行到了 `printf` 所在位置，即 `call printf`，我们通过 `objdump` 反编译 `test` 程序的正文段看看。

```
$ objdump -d -s -j .text test | grep printf
 804837c:       e8 1f ff ff ff          call   80482a0 <printf@p
lt>
```

发现，该地址指向了 `plt` （即过程链接表）即地址 `80482a0` 处。下面查看该地址处的内容。

```
$ objdump -D test | grep "80482a0" | grep -v call
080482a0 <printf@plt>:
 80482a0:       ff 25 8c 95 04 08       jmp    *0x804958c
```

发现 `80482a0` 地址对应的是一条跳转指令，跳转到 `0x804958c` 地址指向的地址。到底 `0x804958c` 地址本身在什么地方呢？我们能否从 `.dynamic` 节区（该节区存放了和动态链接相关的数据）获取相关的信息呢？

```
$ readelf -d test

Dynamic section at offset 0x4ac contains 20 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libc.s
o.6]
 0x0000000c (INIT)                       0x8048258
 0x0000000d (FINI)                       0x8048454
 0x00000004 (HASH)                       0x8048148
 0x00000005 (STRTAB)                     0x80481c0
 0x00000006 (SYMTAB)                     0x8048170
 0x0000000a (STRSZ)                      76 (bytes)
 0x0000000b (SYMENT)                     16 (bytes)
 0x00000015 (DEBUG)                      0x0
 0x00000003 (PLTGOT)                     0x8049578
 0x00000002 (PLTRELSZ)                   24 (bytes)
 0x00000014 (PLTREL)                     REL
 0x00000017 (JMPREL)                     0x8048240
 0x00000011 (REL)                        0x8048238
 0x00000012 (RELSZ)                      8 (bytes)
 0x00000013 (RELENT)                     8 (bytes)
 0x6ffffffe (VERNEED)                    0x8048218
 0x6fffffff (VERNEEDNUM)                 1
 0x6ffffff0 (VERSYM)                     0x804820c
 0x00000000 (NULL)                       0x0
```

发现 `0x8049578` 地址和 `0x804958c` 地址比较近，通过资料 [8] 查到前者正好是 `.got.plt` （即过程链接表）对应的全局偏移表的入口地址。难道 `0x804958c` 正好位于 `.got.plt` 节区中？

全局偏移表（got）

现在进入全局偏移表看看，

```
$ readelf -x .got.plt test

Hex dump of section '.got.plt':
  0x08049578 ac940408 00000000 00000000 86820408 ...............
.
  0x08049588 96820408 a6820408                   ........
```

从上述结果可以看出 `0x804958c` 地址（即 `0x08049588+4` ）处存放的是 `a6820408` ，考虑到我的实验平台是 `i386` ，字节顺序是 `little-endian` 的，所以实际数值应该是 `080482a6` ，也就是说 `*` （ `0x804958c` ） 的值是 080482a6 ，这个地址刚好是过程链接表的最后一项 call 80482a0printf@plt 中 80482a0 地址往后偏移 6 个字节，容易猜到该地址应该就是 jmp` 指令的后一条地址。

```
$ objdump -d -d -s -j .plt test |  grep "080482a0 <printf@plt>:"
 -A 3
080482a0 <printf@plt>:
 80482a0:       ff 25 8c 95 04 08       jmp    *0x804958c
 80482a6:       68 10 00 00 00          push   $0x10
 80482ab:       e9 c0 ff ff ff          jmp    8048270 <_init+0x
18>
```

`80482a6` 地址恰巧是一条 `push` 指令，随后是一条 `jmp` 指令（暂且不管 `push` 指令入栈的内容有什么意义），执行完 `push` 指令之后，就会跳转到 `8048270` 地址处，下面看看 `8048270` 地址处到底有哪些指令。

```
$ objdump -d -d -s -j .plt test | grep -v "jmp    8048270 <_init
+0x18>" | grep "08048270" -A 2
08048270 <__gmon_start__@plt-0x10>:
 8048270:       ff 35 7c 95 04 08       pushl  0x804957c
 8048276:       ff 25 80 95 04 08       jmp    *0x8049580
```

同样是一条入栈指令跟着一条跳转指令。不过这两个地址 `0x804957c` 和 `0x8049580` 是连续的，而且都很熟悉，刚好都在 `.got.plt` 表里头（从上面我们已经知道 `.got.plt` 的入口是 `0x08049578` ）。这样的话，我们得确认这两个地址到底有什么内容。

```
$ readelf -x .got.plt test

Hex dump of section '.got.plt':
  0x08049578 ac940408 00000000 00000000 86820408 ...............
.
  0x08049588 96820408 a6820408                   ........
```

不过，遗憾的是通过 `readelf` 查看到的这两个地址信息都是 0，它们到底是什么呢？

现在只能求助参考资料 [8]，该资料的“3.8.5 过程链接表”部分在介绍过程链接表和全局偏移表相互合作解析符号的过程中的三步涉及到了这两个地址和前面没有说明的 `push` $ `0x10` 指令。

- 在程序第一次创建内存映像时，动态链接器为全局偏移表的第二（`0x804957c`）和第三项（`0x8049580`）设置特殊值。
- 原步骤 5。在跳转到 `08048270 <__gmon_start__@plt-0x10>`，即过程链接表的第一项之前，有一条压入栈指令，即 `push $0x10`，0x10是相对于重定位表起始地址的一个偏移地址，这个偏移地址到底有什么用呢？它应该是提供给动态链接器的什么信息吧？后面再说明。
- 原步骤 6。跳转到过程链接表的第一项之后，压入了全局偏移表中的第二项（即 `0x804957c` 处），“为动态链接器提供了识别信息的机会”（具体是什么呢？后面会简单提到，但这个并不是很重要)，然后跳转到全局偏移表的第三项（`0x8049580`，这一项比较重要），把控制权交给动态链接器。

从这三步发现程序运行时地址 `0x8049580` 处存放的应该是动态链接器的入口地址，而重定位表 `0x10` 位置处和 `0x804957c` 处应该为动态链接器提供了解析符号需要的某些信息。

在继续之前先总结一下过程链接表和全局偏移表。上面的操作过程仅仅从“局部”看过了这两个表，但是并没有宏观地看里头的内容。下面将宏观的分析一下， 对于过程链接表：

```
$ objdump -d -d -s -j .plt test
08048270 <__gmon_start__@plt-0x10>:
 8048270:       ff 35 7c 95 04 08       pushl  0x804957c
 8048276:       ff 25 80 95 04 08       jmp    *0x8049580
 804827c:       00 00                   add    %al,(%eax)
        ...

08048280 <__gmon_start__@plt>:
 8048280:       ff 25 84 95 04 08       jmp    *0x8049584
 8048286:       68 00 00 00 00          push   $0x0
 804828b:       e9 e0 ff ff ff          jmp    8048270 <_init+0x
18>

08048290 <__libc_start_main@plt>:
 8048290:       ff 25 88 95 04 08       jmp    *0x8049588
 8048296:       68 08 00 00 00          push   $0x8
 804829b:       e9 d0 ff ff ff          jmp    8048270 <_init+0x
18>

080482a0 <printf@plt>:
 80482a0:       ff 25 8c 95 04 08       jmp    *0x804958c
 80482a6:       68 10 00 00 00          push   $0x10
 80482ab:       e9 c0 ff ff ff          jmp    8048270 <_init+0x
18>
```

除了该表中的第一项外，其他各项实际上是类似的。而最后一项 `080482a0 <printf@plt>` 和第一项我们都分析过，因此不难理解其他几项的作用。过程链接表没有办法单独工作，因为它和全局偏移表是关联的，所以在说明它的作用之前，先从总体上来看一下全局偏移表。

```
$ readelf -x .got.plt test

Hex dump of section '.got.plt':
  0x08049578 ac940408 00000000 00000000 86820408 ...............
.
  0x08049588 96820408 a6820408                   ........
```

比较全局偏移表中 `0x08049584` 处开始的数据和过程链接表第二项开始的连续三项中 `push` 指定所在的地址，不难发现，它们是对应的。而 `0x0804958c` 即 `push 0x10` 对应的地址我们刚才提到过（下一节会进一步分析），其他几项的作用类似，都是跳回到过程链接表的 `push` 指令处，随后就跳转到过程链接表的第一项，以便解析相应的符号（实际上过程链接表的第一个表项是进入动态链接器，而之前的连续两个指令则传送了需要解析的符号等信息）。另外 `0x08049578` 和 `0x08049580` 处分别存放有传递给动态链接库的相关信息和动态链接器本身的入口地址。但是还有一个地址 `0x08049578`，这个地址刚好是 `.dynamic` 的入口地址，该节区存放了和动态链接过程相关的信息，资料 [8] 提到这个表项实际上保留给动态链接器自己使用的，以便在不依赖其他程序的情况下对自己进行初始化，所以下面将不再关注该表项。

```
$ objdump -D test | grep 080494ac
080494ac <_DYNAMIC>:
```

重定位表

这里主要接着上面的 `push 0x10` 指令来分析。通过资料 [8] 发现重定位表包含如何修改其他节区的信息，以便动态链接器对某些节区内的符号地址进行重定位（修改为新的地址）。那到底重定位表项提供了什么样的信息呢？

- 每一个重定位项有三部分内容，我们重点关注前两部分。
- 第一部分是 `r_offset`，这里考虑的是可执行文件，因此根据资料发现，它的取值是被重定位影响（可以说改变或修改）到的存储单元的虚拟地址。
- 第二部分是 `r_info`，此成员给出要进行重定位的符号表索引（重定位表项引用到的符号表），以及将实施的重定位类型（如何进行符号的重定位）。(Type)。

先来看看重定位表的具体内容，

```
$ readelf -r test

Relocation section '.rel.dyn' at offset 0x238 contains 1 entries
:
 Offset     Info    Type            Sym.Value  Sym. Name
08049574  00000106 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x240 contains 3 entries
:
 Offset     Info    Type            Sym.Value  Sym. Name
08049584  00000107 R_386_JUMP_SLOT   00000000   __gmon_start__
08049588  00000207 R_386_JUMP_SLOT   00000000   __libc_start_mai
n
0804958c  00000407 R_386_JUMP_SLOT   00000000   printf
```

仅仅关注和过程链接表相关的 `.rel.plt` 部分， `0x10` 刚好是 `1*16+0*1` ，即 16 字节，作为重定位表的偏移，刚好对应该表的第三行。发现这个结果中竟然包含了和 `printf` 符号相关的各种信息。不过重定位表中没有直接指定符号 `printf` ，而是根据 `r_info` 部分从动态符号表中计算出来的，注意观察上述结果中的 `Info` 一列的 1，2，4 和下面结果的 `Num` 列的对应关系。

```
$ readelf -s test | grep ".dynsym" -A 6
Symbol table '.dynsym' contains 5 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start_
_
     2: 00000000   410 FUNC    GLOBAL DEFAULT  UND __libc_start_
main@GLIBC_2.0 (2)
     3: 08048474     4 OBJECT  GLOBAL DEFAULT   14 _IO_stdin_use
d
     4: 00000000    57 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_
2.0 (2)
```

也就是说在执行过程链接表中的第一项的跳转指令（ `jmp *0x8049580` ）调用动态链接器以后，动态链接器因为有了 `push 0x10` ，从而可以通过该重定位表项中的 `r_info` 找到对应符号（ `printf` ）在符号表（ `.dynsym` ）中的相关信息。

除此之外，符号表中还有 `Offset` （ `r_offset` ） 以及 Type 这两个重要信息，前者表示该重定位操作后可能影响的地址 0804958c ，这个地址刚好是 got 表项的最后一项，原来存放的是 push 0x10 指令的地址。这意味着，该地址处的内容将被修改，而如何修改呢？根据 Type 类型 R_386_JUMP_SLOT`，通过资料 [8] 查找到该类型对应的说明如下（原资料有误，下面做了修改）：链接编辑器创建这种重定位类型主要是为了支持动态链接。其偏移地址成员给出过程链接表项的位置。动态链接器修改全局偏移表项的内容，把控制传输给指定符号的地址。

这说明，动态链接器将根据该类型对全局偏移表中的最有一项，即 `0804958c` 地址处的内容进行修改，修改为符号的实际地址，即 `printf` 函数在动态链接库的内存映像中的地址。

到这里，动态链接的宏观过程似乎已经了然于心，不过一些细节还是不太清楚。

下面先介绍动态链接库的创建，隐式调用和显示调用，接着进一步澄清上面还不太清楚的细节，即全局偏移表中第二项到底传递给了动态链接器什么信息？第三项是否就是动态链接器的地址？并讨论通过设置 `LD_BIND_NOW` 而不采用默认的 lazy mode 进行动态链接和采用 lazy mode 动态链接的区别？

动态链接库的创建和调用

在介绍动态符号链接的更多细节之前，先来了解一下动态链接库的创建和两种使用方法，进而引出符号解析的后台细节。

创建动态链接库

首先来创建一个简单动态链接库。

代码：

```
/* myprintf.c */
#include <stdio.h>

int myprintf(char *str)
{
        printf("%s\n", str);
        return 0;
}
```

```
/* myprintf.h */
#ifndef _MYPRINTF_H
#define _MYPRINTF_H

int myprintf(char *);

#endif
```

演示：

```
$ gcc -c myprintf.c
$ gcc -shared -Wl,-soname,libmyprintf.so.0 -o libmyprintf.so.0.0
 myprintf.o
$ ln -sf libmyprintf.so.0.0 libmyprintf.so.0
$ ln -fs libmyprintf.so.0 libmyprintf.so
$ ls
libmyprintf.so  libmyprintf.so.0  libmyprintf.so.0.0  myprintf.c
  myprintf.h  myprintf.o
```

得到三个文件 `libmyprintf.so` ，`libmyprintf.so.0` ，`libmyprintf.so.0.0` ，这些库暂且存放在当前目录下。这里有一个问题值得关注，那就是为什么要创建两个符号链接呢？答案是为了在不影响兼容性的前提下升级库 [5] 。

隐式使用该库

现在写一段代码来使用该库，调用其中的 `myprintf` 函数，这里是隐式使用该库：在代码中并没有直接使用该库，而是通过调用 `myprintf` 隐式地使用了该库，在编译引用该库的可执行文件时需要通过 `-l` 参数指定该库的名字。

```
/* test.c */
#include <stdio.h>
#include <myprintf.h>

int main()
{
        myprintf("Hello World");

        return 0;
}
```

编译：

```
$ gcc -o test test.c -lmyprintf -L./ -I./
```

直接运行 `test`，提示找不到该库，因为库的默认搜索路径里头没有包含当前目录：

```
$ ./test
./test: error while loading shared libraries: libmyprintf.so: cannot open shared object file: No such file or directory
```

如果指定库的搜索路径，则可以运行：

```
$ LD_LIBRARY_PATH=$PWD ./test
Hello World
```

显式使用库

`LD_LIBRARY_PATH` 环境变量使得库可以放到某些指定的路径下面，而无须在调用程序中显式的指定该库的绝对路径，这样避免了把程序限制在某些绝对路径下，方便库的移动。

虽然显式调用有不便，但是能够避免隐式调用搜索路径的时间消耗，提高效率，除此之外，显式调用为我们提供了一组函数调用，让符号的重定位过程一览无遗。

```
/* test1.c */

#include <dlfcn.h>      /* dlopen, dlsym, dlerror */
#include <stdlib.h>     /* exit */
#include <stdio.h>      /* printf */

#define LIB_SO_NAME     "./libmyprintf.so"
#define FUNC_NAME "myprintf"

typedef int (*func)(char *);

int main(void)
{
        void *h;
        char *e;
        func f;

        h = dlopen(LIB_SO_NAME, RTLD_LAZY);
        if ( !h ) {
                printf("failed load libary: %s\n", LIB_SO_NAME);
                exit(-1);
        }
        f = dlsym(h, FUNC_NAME);
        e = dlerror();
        if (e != NULL) {
                printf("search %s error: %s\n", FUNC_NAME, LIB_S
O_NAME);
                exit(-1);
        }
        f("Hello World");

        exit(0);
}
```

演示：

```
$ gcc -o test1 test1.c -ldl
```

这种情况下，无须包含头文件。从这个代码中很容易看出符号重定位的过程：

- 首先通过 `dlopen` 找到依赖库，并加载到内存中，再返回该库的 `handle` ，通过 `dlopen` 我们可以指定 `RTLD_LAZY` 采用 `lazy mode` 动态链接模式，如果采用 `RTLD_NOW` 则和隐式调用时设置 `LD_BIN_NOW` 类似。
- 找到该库以后就是对某个符号进行重定位，这里是确定 `myprintf` 函数的地址。
- 找到函数地址以后就可以直接调用该函数了。

关于 `dlopen` ， `dlsym` 等后台工作细节建议参考资料 [15] 。

隐式调用的动态符号链接过程和上面类似。下面通过一些实例来确定之前没有明确的两个内容：即全局偏移表中的第二项和第三项，并进一步讨论lazy mode和非lazy mode的区别。

动态链接过程

因为通过 `ELF` 文件，我们就可以确定全局偏移表的位置，因此为了确定全局偏移表位置的第三项和第四项的内容，有两种办法：

- 通过 `gdb` 调试。
- 直接在函数内部打印。

因为资料[3]详细介绍了第一种方法，这里试着通过第二种方法来确定这两个地址的值。

```
/**
 * got.c -- get the relative content of the got(global offset ta
ble) of an elf file
 */

#include <stdio.h>

#define GOT 0x8049614

int main(int argc, char *argv[])
{
        long got2, got3;
        long old_addr, new_addr;

        got2=*(long *)(GOT+4);
        got3=*(long *)(GOT+8);
        old_addr=*(long *)(GOT+24);

        printf("Hello World\n");

        new_addr=*(long *)(GOT+24);

        printf("got2: 0x%0x, got3: 0x%0x, old_addr: 0x%0x, new_a
ddr: 0x%0x\n",
                                        got2, got3, old_addr, ne
w_addr);

        return 0;
}
```

在写好上面的代码后就需要确定全局偏移表的地址，然后把该地址设置为代码中的宏 `GOT` 。

```
$ make got
$ readelf -d got | grep PLTGOT
 0x00000003 (PLTGOT)                        0x8049614
```

注：这里假设大家用的都是 `i386` 的系统，如果要在 `X86_64` 位系统上要编译生成 `i386` 上的可执行文件，需要给 `gcc` 传递一个 `-m32` 参数，例如：

```
$ gcc -m32 -o got got.c
```

把地址 `0x8049614` 替换到上述代码中，然后重新编译运行，查看结果。

```
$ make got
$ Hello World
got2: 0xb7f376d8, got3: 0xb7f2ef10, old_addr: 0x80482da, new_add
r: 0xb7e19a20
$ ./got
Hello World
got2: 0xb7f1e6d8, got3: 0xb7f15f10, old_addr: 0x80482da, new_add
r: 0xb7e00a20
```

通过两次运行，发现全局偏移表中的这两项是变化的，并且 `printf` 的地址对应的 `new_addr` 也是变化的，说明 `libc` 和 `ld-linux` 这两个库启动以后对应的虚拟地址并不确定。因此，无法直接跟踪到那个地址处的内容，还得借助调试工具，以便确认它们。

下面重新编译 `got`，加上 `-g` 参数以便调试，并通过调试确认 `got2`，`got3`，以及调用 `printf` 前后 `printf` 地址的重定位情况。

```
$ gcc -g -o got got.c
$ gdb -q ./got
(gdb) l
5       #include <stdio.h>
6
7       #define GOT 0x8049614
8
9       int main(int argc, char *argv[])
10      {
11              long got2, got3;
12              long old_addr, new_addr;
13
14              got2=*(long *)(GOT+4);
(gdb) l
15              got3=*(long *)(GOT+8);
16              old_addr=*(long *)(GOT+24);
17
18              printf("Hello World\n");
19
20              new_addr=*(long *)(GOT+24);
21
22              printf("got2: 0x%0x, got3: 0x%0x, old_addr: 0x%0x, new_addr: 0x%0x\n",
23                                              got2, got3, old_addr, new_addr);
24
```

在第一个 `printf` 处设置一个断点：

```
(gdb) break 18
Breakpoint 1 at 0x80483c3: file got.c, line 18.
```

在第二个 `printf` 处设置一个断点：

```
(gdb) break 22
Breakpoint 2 at 0x80483dd: file got.c, line 22.
```

运行到第一个 `printf` 之前会停止：

```
(gdb) r
Starting program: /mnt/hda8/Temp/c/program/got

Breakpoint 1, main () at got.c:18
18              printf("Hello World\n");
```

查看执行 `printf` 之前的全局偏移表内容：

```
(gdb) x/8x 0x8049614
0x8049614 <_GLOBAL_OFFSET_TABLE_>:      0x08049548      0xb7f3c6
d8      0xb7f33f10      0x080482aa
0x8049624 <_GLOBAL_OFFSET_TABLE_+16>:   0xb7ddbd20      0x080482
ca      0x080482da      0x00000000
```

查看 `GOT` 表项的最有一项，发现刚好是 `PLT` 表中 `push` 指令的地址：

```
(gdb) disassemble 0x080482da
Dump of assembler code for function puts@plt:
0x080482d4 <puts@plt+0>:        jmp    *0x804962c
0x080482da <puts@plt+6>:        push   $0x18
0x080482df <puts@plt+11>:       jmp    0x8048294 <_init+24>
```

说明此时还没有进行进行符号的重定位，不过发现并非 `printf`，而是 `puts(1)`。

接着查看 `GOT` 第三项的内容，刚好是 `dl-linux` 对应的代码：

```
(gdb) disassemble 0xb7f33f10
Dump of assembler code for function _dl_runtime_resolve:
0xb7f33f10 <_dl_runtime_resolve+0>:     push   %eax
0xb7f33f11 <_dl_runtime_resolve+1>:     push   %ecx
0xb7f33f12 <_dl_runtime_resolve+2>:     push   %edx
```

可通过 `nm /lib/ld-linux.so.2 | grep _dl_runtime_resolve` 进行确认。

然后查看 `GOT` 表第二项处的内容，看不出什么特别的信息，反编译时提示无法反编译：

```
(gdb) x/8x 0xb7f3c6d8
0xb7f3c6d8:     0x00000000      0xb7f39c3d      0x08049548
0xb7f3c9b8
0xb7f3c6e8:     0x00000000      0xb7f3c6d8      0x00000000
0xb7f3c9a4
```

在 `*(0xb7f33f10)` 指向的代码处设置一个断点，确认它是否被执行：

```
(gdb) break *(0xb7f33f10)
break *(0xb7f33f10)
Breakpoint 3 at 0xb7f3cf10
(gdb) c
Continuing.

Breakpoint 3, 0xb7f3cf10 in _dl_runtime_resolve () from /lib/ld-
linux.so.2
```

继续运行，直到第二次调用 `printf` ：

```
(gdb)  c
Continuing.
Hello World

Breakpoint 2, main () at got.c:22
22              printf("got2: 0x%0x, got3: 0x%0x, old_addr: 0x%0
x, new_addr: 0x%0x\n",
```

再次查看 `GOT` 表项，发现 `GOT` 表的最后一项的值应该被修改：

```
(gdb) x/8x 0x8049614
0x8049614 <_GLOBAL_OFFSET_TABLE_>:      0x08049548      0xb7f3c6
d8      0xb7f33f10      0x080482aa
0x8049624 <_GLOBAL_OFFSET_TABLE_+16>:   0xb7ddbd20      0x080482
ca      0xb7e1ea20      0x00000000
```

查看 `GOT` 表最后一项，发现变成了 `puts` 函数的代码，说明进行了符号 `puts` 的重定位（2）：

```
(gdb) disassemble 0xb7e1ea20
Dump of assembler code for function puts:
0xb7e1ea20 <puts+0>:    push   %ebp
0xb7e1ea21 <puts+1>:    mov    %esp,%ebp
0xb7e1ea23 <puts+3>:    sub    $0x1c,%esp
```

通过演示发现一个问题（1）（2），即本来调用的是 `printf`，为什么会进行 `puts` 的重定位呢？通过 `gcc -S` 参数编译生成汇编代码后发现，`gcc` 把 `printf` 替换成了 `puts`，因此不难理解程序运行过程为什么对 `puts` 进行了重定位。

从演示中不难发现，当符号被使用到时才进行重定位。因为通过调试发现在执行 `printf` 之后，`GOT` 表项的最后一项才被修改为 `printf`（确切的说是 `puts`）的地址。这就是所谓的 `lazy mode` 动态符号链接方式。

除此之外，我们容易发现 `GOT` 表第三项确实是 `ld-linux.so` 中的某个函数地址，并且发现在执行 `printf` 语句之前，先进入了 `ld-linux.so` 的 `_dl_runtime_resolve` 函数，而且在它返回之后，`GOT` 表的最后一项才变为 `printf`（`puts`）的地址。

本来打算通过第一个断点确认第二次调用 `printf` 时不再需要进行动态符号链接的，不过因为 `gcc` 把第一个替换成了 `puts`，所以这里没有办法继续调试。如果想确认这个，你可以通过写两个一样的 `printf` 语句看看。实际上第一次链接以后，`GOT` 表的第三项已经修改了，当下次再进入过程链接表，并执行 `jmp *(全局偏移表中某一个地址)` 指令时，`*(全局偏移表中某一个地址)` 已经被修改为了对应符号的实际地址，这样 `jmp` 语句会自动跳转到符号的地址处运行，执行具体的函数代码，因此无须再进行重定位。

到现在 `GOT` 表中只剩下第二项还没有被确认，通过资料 [3] 我们发现，该项指向一个 `link_map` 类型的数据，是一个鉴别信息，具体作用对我们来说并不是很重要，如果想了解，请参考资料 [16]。

下面通过设置 `LD_BIND_NOW` 再运行一下 `got` 程序并查看结果，比较它与默认的动态链接方式（`lazy mode`）的异同。

- 设置 `LD_BIND_NOW` 环境变量的运行结果

```
$ LD_BIND_NOW=1 ./got
Hello World
got2: 0x0, got3: 0x0, old_addr: 0xb7e61a20, new_addr: 0xb7
e61a20
```

- 默认情况下的运行结果

```
$ ./got
Hello World
got2: 0xb7f806d8, got3: 0xb7f77f10, old_addr: 0x80482da, n
ew_addr: 0xb7e62a20
```

通过比较容易发现，在非 `lazy mode` （设置 `LD_BIND_NOW` 后）下，程序运行之前符号的地址就已经被确定，即调用 `printf` 之前 `GOT` 表的最后一项已经被确定为了 `printf` 函数对应的地址，即 `0xb7e61a20` ，因此在程序运行之后， `GOT` 表的第二项和第三项就保持为 0，因为此时不再需要它们进行符号的重定位了。通过这样一个比较，就更容易理解 `lazy mode` 的特点了：在用到的时候才解析。

到这里，符号动态链接的细节基本上就已经清楚了。

参考资料

- LINUX 系统中动态链接库的创建与使用
- LINUX 动态链接库高级应用
- ELF 动态解析符号过程(修订版)
- 如何在 Linux 下调试动态链接库
- Dissecting shared libraries
- 关于 Linux 和 Unix 动态链接库的安全
- Linux 系统下解析 ELF 文件 DT_RPATH 后门
- ELF 文件格式分析

- 缓冲区溢出与注入分析(第二部分：缓冲区溢出和注入实例)
- Gcc 编译的背后(第二部分：汇编和链接)
- 程序执行的那一刹那
- What is Linux-gate.so.1: [1], [2], [3]
- Linux 下缓冲区溢出攻击的原理及对策
- Intel 平台下 Linux 中 ELF 文件动态链接的加载、解析及实例分析 part1, part2
- ELF file format and ABI

缓冲区溢出与注入分析

前言

虽然程序加载以及动态符号链接都已经很理解了，但是这伙却被进程的内存映像给"纠缠"住。看着看着就一发不可收拾——很有趣。

下面一起来探究“缓冲区溢出和注入”问题（主要是关心程序的内存映像）。

进程的内存映像

永远的 `Hello World` ，太熟悉了吧，

```
#include <stdio.h>
int main(void)
{
    printf("Hello World\n");
    return 0;
}
```

如果要用内联汇编（ `inline assembly` ）来写呢？

```
 1  /* shellcode.c */
 2  void main()
 3  {
 4      __asm__ __volatile__("jmp forward;"
 5                  "backward:"
 6                          "popl   %esi;"
 7                          "movl   $4, %eax;"
 8                          "movl   $2, %ebx;"
 9                          "movl   %esi, %ecx;"
10                          "movl   $12, %edx;"
11                          "int    $0x80;"    /* system call 1 */
12                          "movl   $1, %eax;"
13                          "movl   $0, %ebx;"
14                          "int    $0x80;"    /* system call 2 */
15                  "forward:"
16                          "call   backward;"
17                          ".string \"Hello World\\n\";");
18  }
```

看起来很复杂，实际上就做了一个事情，往终端上写了个 `Hello World` 。不过这个非常有意思。先简单分析一下流程：

- 第 4 行指令的作用是跳转到第 15 行（即 `forward` 标记处），接着执行第 16 行。
- 第 16 行调用 `backward` ，跳转到第 5 行，接着执行 6 到 14 行。

- 第 6 行到第 11 行负责在终端打印出 `Hello World` 字符串（等一下详细介绍）。
- 第 12 行到第 14 行退出程序（等一下详细介绍）。

为了更好的理解上面的代码和后续的分析，先来介绍几个比较重要的内容。

常用寄存器初识

`X86` 处理器平台有三个常用寄存器：程序指令指针、程序堆栈指针与程序基指针：

寄存器	名称	注释
EIP	程序指令指针	通常指向下一条指令的位置
ESP	程序堆栈指针	通常指向当前堆栈的当前位置
EBP	程序基指针	通常指向函数使用的堆栈顶端

当然，上面都是扩展的寄存器，用于 32 位系统，对应的 16 系统为 `ip` ，`sp` ，`bp` 。

call，ret 指令的作用分析

- `call` 指令

  跳转到某个位置，并在之前把下一条指令的地址（ `EIP` ）入栈（为了方便”程序“返回以后能够接着执行）。这样的话就有：

  ```
  call backward   ==>   push eip
                        jmp backward
  ```

- `ret` 指令

  通常 `call` 指令和 `ret` 是配合使用的，前者压入跳转前的下一条指令地址，后者弹出 `call` 指令压入的那条指令，从而可以在函数调用结束以后接着执行后面的指令。

  ```
  ret                    ==>   pop eip
  ```

通常在函数调用后，还需要恢复 `esp` 和 `ebp`，恢复 `esp` 即恢复当前栈指针，以便释放调用函数时为存储函数的局部变量而自动分配的空间；恢复 `ebp` 是从栈中弹出一个数据项（通常函数调用过后的第一条语句就是 `push ebp`），从而恢复当前的函数指针为函数调用者本身。这两个动作可以通过一条 `leave` 指令完成。

这三个指令对我们后续的解释会很有帮助。更多关于 Intel 的指令集，请参考：Intel 386 Manual, x86 Assembly Language FAQ：part1, part2, part3.

什么是系统调用（以 Linux 2.6.21 版本和 x86 平台为例）

系统调用是用户和内核之间的接口，用户如果想写程序，很多时候直接调用了 C 库，并没有关心系统调用，而实际上 C 库也是基于系统调用的。这样应用程序和内核之间就可以通过系统调用联系起来。它们分别处于操作系统的用户空间和内核空间（主要是内存地址空间的隔离）。

```
用户空间          应用程序(Applications)
                           |         |
                           |      C库（如glibc）
                           |         |
                          系统调用(System Calls，如sys_read, sys_writ
e, sys_exit)
                               |
内核空间              内核(Kernel)
```

系统调用实际上也是一些函数，它们被定义在 `arch/i386/kernel/sys_i386.c`（老的在 `arch/i386/kernel/sys.c`）文件中，并且通过一张系统调用表组织，该表在内核启动时就已经加载了，这个表的入口在内核源代码的 `arch/i386/kernel/syscall_table.S` 里头（老的在 `arch/i386/kernel/entry.S`）。这样，如果想添加一个新的系统调用，修改上面两个内核中的文件，并重新编译内核就可以。当然，如果要在应用程序中使用它们，还得把它写到 `include/asm/unistd.h` 中。

如果要在 C 语言中使用某个系统调用，需要包含头文件 `/usr/include/asm/unistd.h`，里头有各个系统调用的声明以及系统调用号（对应于调用表的入口，即在调用表中的索引，为方便查找调用表而设立的）。如

果是自己定义的新系统调用，可能还要在开头用宏 `_syscall(type, name, type1, name1...)` 来声明好参数。

如果要在汇编语言中使用，需要用到 `int 0x80` 调用，这个是系统调用的中断入口。涉及到传送参数的寄存器有这么几个， `eax` 是系统调用号（可以到 `/usr/include/asm-i386/unistd.h` 或者直接到 `arch/i386/kernel/syscall_table.S` 查到），其他寄存器如 `ebx` ， `ecx` ， `edx` ， `esi` ， `edi` 一次存放系统调用的参数。而系统调用的返回值存放在 `eax` 寄存器中。

下面我们就很容易解释前面的 `Shellcode.c` 程序流程的 2，3 两部分了。因为都用了 `int 0x80` 中断，所以都用到了系统调用。

第 3 部分很简单，用到的系统调用号是 1，通过查表（查 `/usr/include/asm-i386/unistd.h` 或 `arch/i386/kernel/syscall_table.S` ）可以发现这里是 `sys_exit` 调用，再从 `/usr/include/unistd.h` 文件看这个系统调用的声明，发现参数 `ebx` 是程序退出状态。

第 2 部分比较有趣，而且复杂一点。我们依次来看各个寄存器，首先根据 `eax` 为 4 确定（同样查表）系统调用为 `sys_write` ，而查看它的声明（从 `/usr/include/unistd.h` ），我们找到了参数依次为文件描述符、字符串指针和字符串长度。

- 第一个参数是 `ebx` ，正好是 2，即标准错误输出，默认为终端。
- 第二个参数是 `ecx` ，而 `ecx` 的内容来自 `esi` ， `esi` 来自刚弹出栈的值（见第 6 行 `popl %esi;` ），而之前刚好有 `call` 指令引起了最近一次压栈操作，入栈的内容刚好是 `call` 指令的下一条指令的地址，即 `.string` 所在行的地址，这样 `ecx` 刚好引用了 `Hello World\\n` 字符串的地址。
- 第三个参数是 `edx` ，刚好是 12，即 `Hello World\\n` 字符串的长度（包括一个空字符）。这样， `Shellcode.c` 的执行流程就很清楚了，第 4，5，15，16 行指令的巧妙之处也就容易理解了（把 `.string` 存放在 `call` 指令之后，并用 `popl` 指令把 `eip` 弹出当作字符串的入口）。

什么是 ELF 文件

这里的 ELF 不是“精灵”，而是 Executable and Linking Format 文件，是 Linux 下用来做目标文件、可执行文件和共享库的一种文件格式，它有专门的标准，例如：X86 ELF format and ABI，中文版。

下面简单描述 `ELF` 的格式。

`ELF` 文件主要有三种，分别是：

- 可重定位的目标文件，在编译时用 `gcc` 的 `-c` 参数时产生。
- 可执行文件，这类文件就是我们后面要讨论的可以执行的文件。
- 共享库，这里主要是动态共享库，而静态共享库则是可重定位的目标文件通过 `ar` 命令组织的。

`ELF` 文件的大体结构：

```
ELF Header                 #程序头，有该文件的Magic number(参考man magic)，类型等
Program Header Table       #对可执行文件和共享库有效，它描述下面各个节(section)组成的段
Section1
Section2
Section3
.....
Program Section Table   #仅对可重定位目标文件和静态库有效，用于描述各个Section的重定位信息等。
```

对于可执行文件，文件最后的 `Program Section Table` （节区表）和一些非重定位的 `Section` ，比如 `.comment` ， `.note.XXX.debug` 等信息都可以删除掉，不过如果用 `strip` ， `objcopy` 等工具删除掉以后，就不可恢复了。因为这些信息对程序的运行一般没有任何用处。

`ELF` 文件的主要节区（ `section` ）有 `.data` ， `.text` ， `.bss` ， `.interp` 等，而主要段（ `segment` ）有 `LOAD` ， `INTERP` 等。它们之间（节区和段）的主要对应关系如下：

Section	解释	实例
.data	初始化的数据	比如 `int a=10`
.bss	未初始化的数据	比如 `char sum[100];` 这个在程序执行之前，内核将初始化为 0
.text	程序代码正文	即可执行指令集
.interp	描述程序需要的解释器（动态连接和装载程序）	存有解释器的全路径，如 `/lib/ld-linux.so`

而程序在执行以后，`.data`，`.bss`，`.text` 等一些节区会被 `Program header table` 映射到 `LOAD` 段，`.interp` 则被映射到了 `INTERP` 段。

对于 `ELF` 文件的分析，建议使用 `file`，`size`，`readelf`，`objdump`，`strip`，`objcopy`，`gdb`，`nm` 等工具。

这里简单地演示这几个工具：

```
$ gcc -g -o shellcode shellcode.c  #如果要用gdb调试，编译时加上-g是必须的
shellcode.c: In function ‘main’:
shellcode.c:3: warning: return type of ‘main’ is not ‘int’
f$ file shellcode  #file命令查看文件类型，想了解工作原理，可man magic,man file
shellcode: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), not stripped
$ readelf -l shellcode  #列出ELF文件前面的program head table，后面是它描
                       #述了各个段(segment)和节区(section)的关系,即各个段包含哪些节区。
Elf file type is EXEC (Executable file)
Entry point 0x8048280
There are 7 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000e0 0x000e0 R E 0x4
  INTERP         0x000114 0x08048114 0x08048114 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x0044c 0x0044c R E 0x1000
  LOAD           0x00044c 0x0804944c 0x0804944c 0x00100 0x00104 RW  0x1000
  DYNAMIC        0x000460 0x08049460 0x08049460 0x000c8 0x000c8 RW  0x4
```

```
  NOTE           0x000128 0x08048128 0x08048128 0x00020 0x00020
R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000
RW  0x4

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.versi
on .gnu.version_r
          .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_f
rame
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag
   06
$ size shellcode   #可用size命令查看各个段（对应后面将分析的进程内存映像
）的大小
   text    data     bss     dec     hex filename
    815     256       4    1075     433 shellcode
$ strip -R .note.ABI-tag shellcode #可用strip来给可执行文件“减肥”，
删除无用信息
$ size shellcode              #“减肥”后效果“明显”，对于嵌入式系统应该
有很大的作用
   text    data     bss     dec     hex filename
    783     256       4    1043     413 shellcode
$ objdump -s -j .interp shellcode #这个主要工作是反编译，不过用来查看
各个节区也很厉害

shellcode:     file format elf32-i386

Contents of section .interp:
 8048114 2f6c6962 2f6c642d 6c696e75 782e736f  /lib/ld-linux.so
 8048124 2e3200                               .2.
```

补充：如果要删除可执行文件的 `Program Section Table` ，可以用 A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux 一文的作者写的 elf kicker 工具链中的 `sstrip` 工具。

程序执行基本过程

在命令行下，敲入程序的名字或者是全路径，然后按下回车就可以启动程序，这个具体是怎么工作的呢？

首先要再认识一下我们的命令行，命令行是内核和用户之间的接口，它本身也是一个程序。在 Linux 系统启动以后会为每个终端用户建立一个进程执行一个 Shell 解释程序，这个程序解释并执行用户输入的命令，以实现用户和内核之间的接口。这类解释程序有哪些呢？目前 Linux 下比较常用的有 `/bin/bash` 。那么该程序接收并执行命令的过程是怎么样的呢？

先简单描述一下这个过程：

- 读取用户由键盘输入的命令行。
- 分析命令，以命令名作为文件名，并将其它参数改为系统调用 `execve` 内部处理所要求的形式。
- 终端进程调用 `fork` 建立一个子进程。
- 终端进程本身用系统调用 `wait4` 来等待子进程完成（如果是后台命令，则不等待）。当子进程运行时调用 `execve` ，子进程根据文件名（即命令名）到目录中查找有关文件（这是命令解释程序构成的文件），将它调入内存，执行这个程序（解释这条命令）。
- 如果命令末尾有 `&` 号（后台命令符号），则终端进程不用系统调用 `wait4` 等待，立即发提示符，让用户输入下一个命令，转 1）。如果命令末尾没有 `&` 号，则终端进程要一直等待，当子进程（即运行命令的进程）完成处理后终止，向父进程（终端进程）报告，此时终端进程醒来，在做必要的判别等工作后，终端进程发提示符，让用户输入新的命令，重复上述处理过程。

现在用 `strace` 来跟踪一下程序执行过程中用到的系统调用。

```
$ strace -f -o strace.out test
$ cat strace.out | grep \(.*\) | sed -e "s#[0-9]* \([a-zA-Z0-9_]
*\)(.*).*#\1#g"
execve
brk
access
open
fstat64
mmap2
close
open
read
fstat64
mmap2
mmap2
mmap2
mmap2
close
mmap2
set_thread_area
mprotect
munmap
brk
brk
open
fstat64
mmap2
close
close
close
exit_group
```

相关的系统调用基本体现了上面的执行过程，需要注意的是，里头还涉及到内存映射（ `mmap2` ）等。

下面再罗嗦一些比较有意思的内容，参考《深入理解 Linux 内核》的程序的执行（P681）。

Linux 支持很多不同的可执行文件格式，这些不同的格式是如何解释的呢？平时我们在命令行下敲入一个命令就完了，也没有去管这些细节。实际上 Linux 下有一个 `struct linux_binfmt` 结构来管理不同的可执行文件类型，这个结构中有对应的可执行文件的处理函数。大概的过程如下：

- 在用户态执行了 `execve` 后，引发 `int 0x80` 中断，进入内核态，执行内核态的相应函数 `do_sys_execve` ，该函数又调用 `do_execve` 函数。`do_execve` 函数读入可执行文件，检查权限，如果没问题，继续读入可执行文件需要的相关信息（ `struct linux_binprm` 描述的）。
- 接着执行 `search_binary_handler` ，根据可执行文件的类型（由上一步的最后确定），在 `linux_binfmt` 结构链表（ `formats` ，这个链表可以通过 `register_binfmt` 和 `unregister_binfmt` 注册和删除某些可执行文件的信息，因此注册新的可执行文件成为可能，后面再介绍）上查找，找到相应的结构，然后执行相应的 `load_binary` 函数开始加载可执行文件。在该链表的最后一个元素总是对解释脚本（ `interpreted script` ）的可执行文件格式进行描述的一个对象。这种格式只定义了 `load_binary` 方法，其相应的 `load_script` 函数检查这种可执行文件是否以两个 `#!` 字符开始，如果是，这个函数就以另一个可执行文件的路径名作为参数解释第一行的其余部分，并把脚本文件名作为参数传递以执行这个脚本（实际上脚本程序把自身的内容当作一个参数传递给了解释程序（如 `/bin/bash` ），而这个解释程序通常在脚本文件的开头用 `#!` 标记，如果没有标记，那么默认解释程序为当前 `SHELL` ）。
- 对于 `ELF` 类型文件，其处理函数是 `load_elf_binary` ，它先读入 `ELF` 文件的头部，根据头部信息读入各种数据，再次扫描程序段描述表（ `Program Header Table` ），找到类型为 `PT_LOAD` 的段（即 `.text` ， `.data` ， `.bss` 等节区），将其映射（ `elf_map` ）到内存的固定地址上，如果没有动态连接器的描述段，把返回的入口地址设置成应用程序入口。完成这个功能的是 `start_thread` ，它不启动一个线程，而只是用来修改了 `pt_regs` 中保存的 `PC` 等寄存器的值，使其指向加载的应用程序的入口。当内核操作结束，返回用户态时接着就执行应用程序本身了。
- 如果应用程序使用了动态连接库，内核除了加载指定的可执行文件外，还要把控制权交给动态连接器（ `ld-linux.so` ）以便处理动态连接的程序。内核搜寻段表（ `Program Header Table` ），找到标记为 `PT_INTERP` 段中所对应的动态连接器的名称，并使用 `load_elf_interp` 加载其映像，并把返回的入口地址设置成 `load_elf_interp` 的返回值，即动态链接器的入口。当

> `execve` 系统调用退出时，动态连接器接着运行，它检查应用程序对共享链接库的依赖性，并在需要时对其加载，对程序的外部引用进行重定位（具体过程见《进程和进程的基本操作》）。然后把控制权交给应用程序，从 `ELF` 文件头部中定义的程序进入点（用 `readelf -h` 可以出看到，`Entry point address` 即是）开始执行。（不过对于非 `LIB_BIND_NOW` 的共享库装载是在有外部引用请求时才执行的）。

对于内核态的函数调用过程，没有办法通过 `strace`（它只能跟踪到系统调用层）来做的，因此要想跟踪内核中各个系统调用的执行细节，需要用其他工具。比如可以通过 Ftrace 来跟踪内核具体调用了哪些函数。当然，也可以通过 `ctags/cscope/LXR` 等工具分析内核的源代码。

Linux 允许自己注册我们自己定义的可执行格式，主要接口是 `/procy/sys/fs/binfmt_misc/register`，可以往里头写入特定格式的字符串来实现。该字符串格式如下：`:name:type:offset:string:mask:interpreter:`

- `name` 新格式的标示符
- `type` 识别类型（`M` 表示魔数，`E` 表示扩展）
- `offset` 魔数（`magic number`，请参考 `man magic` 和 `man file`）在文件中的启始偏移量
- `string` 以魔数或者以扩展名匹配的字节序列
- `mask` 用来屏蔽掉 `string` 的一些位
- `interpreter` 程序解释器的完整路径名

Linux 下程序的内存映像

Linux 下是如何给进程分配内存（这里仅讨论虚拟内存的分配）的呢？可以从 `/proc/<pid>/maps` 文件中看到个大概。这里的 `pid` 是进程号。

`/proc` 下有一个文件比较特殊，是 `self`，它链接到当前进程的进程号，例如：

```
$ ls /proc/self -l
lrwxrwxrwx 1 root root 64 2000-01-10 18:26 /proc/self -> 11291/
$ ls /proc/self -l
lrwxrwxrwx 1 root root 64 2000-01-10 18:26 /proc/self -> 11292/
```

看到没？每次都不一样，这样我们通过 `cat /proc/self/maps` 就可以看到 `cat` 程序执行时的内存映像了。

```
$ cat -n /proc/self/maps
     1  08048000-0804c000 r-xp 00000000 03:01 273716     /bin/ca
t
     2  0804c000-0804d000 rw-p 00003000 03:01 273716     /bin/ca
t
     3  0804d000-0806e000 rw-p 0804d000 00:00 0          [heap]
     4  b7b90000-b7d90000 r--p 00000000 03:01 87528      /usr/li
b/locale/locale-archive
     5  b7d90000-b7d91000 rw-p b7d90000 00:00 0
     6  b7d91000-b7ecd000 r-xp 00000000 03:01 466875     /lib/li
bc-2.5.so
     7  b7ecd000-b7ece000 r--p 0013c000 03:01 466875     /lib/li
bc-2.5.so
     8  b7ece000-b7ed0000 rw-p 0013d000 03:01 466875     /lib/li
bc-2.5.so
     9  b7ed0000-b7ed4000 rw-p b7ed0000 00:00 0
    10  b7eeb000-b7f06000 r-xp 00000000 03:01 402817     /lib/ld
-2.5.so
    11  b7f06000-b7f08000 rw-p 0001b000 03:01 402817     /lib/ld
-2.5.so
    12  bfbe3000-bfbf8000 rw-p bfbe3000 00:00 0          [stack]
    13  ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]
```

编号是原文件里头没有的，为了说明方便，用 `-n` 参数加上去的。我们从中可以得到如下信息：

- 第 1，2 行对应的内存区是我们的程序（包括指令，数据等）
- 第 3 到 12 行对应的内存区是堆栈段，里头也映像了程序引用的动态连接库
- 第 13 行是内核空间

总结一下：

- 前两部分是用户空间，可以从 `0x00000000` 到 `0xbfffffff` （在测试的 `2.6.21.5-smp` 上只到 `bfbf8000` ），而内核空间从 `0xC0000000` 到 `0xffffffff` ，分别是 `3G` 和 `1G` ，所以对于每一个进程来说，共占用 `4G` 的虚拟内存空间

- 从程序本身占用的内存，到堆栈段（动态获取内存或者是函数运行过程中用来存储局部变量、参数的空间，前者是 `heap`，后者是 `stack`），再到内核空间，地址是从低到高的
- 栈顶并非 `0xC0000000` 下的一个固定数值

结合相关资料，可以得到这么一个比较详细的进程内存映像表（以 `Linux 2.6.21.5-smp` 为例）：

地址	内核空间	描述
0xC0000000		
	(program flie) 程序名	execve 的第一个参数
	(environment) 环境变量	execve 的第三个参数，main 的第三个参数
	(arguments) 参数	execve 的第二个参数，main 的形参
	(stack) 栈	自动变量以及每次函数调用时所需保存的信息都
		存放在此，包括函数返回地址、调用者的
		环境信息等，函数的参数，局部变量都存放在此
	(shared memory) 共享内存	共享内存的大概位置
	...	
	...	
	(heap) 堆	主要在这里进行动态存储分配，比如 malloc，new 等。
	...	
	.bss (uninitilized data)	没有初始化的数据（全局变量哦）
	.data (initilized global data)	已经初始化的全局数据（全局变量）
	.text (Executable Instructions)	通常是可执行指令
0x08048000		
0x00000000		...

光看没有任何概念，我们用 `gdb` 来看看刚才那个简单的程序。

```
$ gcc -g -o shellcode shellcode.c   #要用gdb调试，在编译时需要加-g参数
$ gdb -q ./shellcode
(gdb) set args arg1 arg2 arg3 arg4  #为了测试，设置几个参数
```

```
(gdb) l                             #浏览代码
1 /* shellcode.c */
2 void main()
3 {
4       __asm__ __volatile__("jmp forward;"
5       "backward:"
6          "popl   %esi;"
7          "movl   $4, %eax;"
8          "movl   $2, %ebx;"
9          "movl   %esi, %ecx;"
10                 "movl   $12, %edx;"
(gdb) break 4                 #在汇编入口设置一个断点，让程序运行后停到这里
Breakpoint 1 at 0x8048332: file shellcode.c, line 4.
(gdb) r                       #运行程序
Starting program: /mnt/hda8/Temp/c/program/shellcode arg1 arg2 arg3 arg4

Breakpoint 1, main () at shellcode.c:4
4       __asm__ __volatile__("jmp forward;"
(gdb) print $esp              #打印当前堆栈指针值，用于查找整个栈的栈顶
$1 = (void *) 0xbffe1584
(gdb) x/100s $esp+4000        #改变后面的4000，不断往更大的空间找
(gdb) x/1s 0xbffe1fd9         #在 0xbffe1fd9 找到了程序名，这里是该次运行时的栈顶
0xbffe1fd9:      "/mnt/hda8/Temp/c/program/shellcode"
(gdb) x/10s 0xbffe17b7        #其他环境变量信息
0xbffe17b7:      "CPLUS_INCLUDE_PATH=/usr/lib/qt/include"
0xbffe17de:      "MANPATH=/usr/local/man:/usr/man:/usr/X11R6/man:/usr/lib/java/man:/usr/share/texmf/man"
0xbffe1834:      "HOSTNAME=falcon.lzu.edu.cn"
0xbffe184f:      "TERM=xterm"
0xbffe185a:      "SSH_CLIENT=219.246.50.235 3099 22"
0xbffe187c:      "QTDIR=/usr/lib/qt"
0xbffe188e:      "SSH_TTY=/dev/pts/0"
0xbffe18a1:      "USER=falcon"
...
(gdb) x/5s 0xbffe1780    #一些传递给main函数的参数，包括文件名和其他参数
0xbffe1780:      "/mnt/hda8/Temp/c/program/shellcode"
```

```
0xbffe17a3:      "arg1"
0xbffe17a8:      "arg2"
0xbffe17ad:      "arg3"
0xbffe17b2:      "arg4"
(gdb) print init  #打印init函数的地址，这个是/usr/lib/crti.o里头的函数，做一些初始化操作
$2 = {<text variable, no debug info>} 0xb7e73d00 <init>
(gdb) print fini   #也在/usr/lib/crti.o中定义，在程序结束时做一些处理工作
$3 = {<text variable, no debug info>} 0xb7f4a380 <fini>
(gdb) print _start #在/usr/lib/crt1.o，这个才是程序的入口，必须的，ld会检查这个
$4 = {<text variable, no debug info>} 0x8048280 <__libc_start_main@plt+20>
(gdb) print main   #这里是我们的main函数
$5 = {void ()} 0x8048324 <main>
```

补充：在进程的内存映像中可能看到诸如 `init` ， `fini` ， `_start` 等函数（或者是入口），这些东西并不是我们自己写的啊？为什么会跑到我们的代码里头呢？实际上这些东西是链接的时候 `gcc` 默认给连接进去的，主要用来做一些进程的初始化和终止的动作。更多相关的细节可以参考资料如何获取当前进程之静态影像文件和"The Linux Kernel Primer"， P234， Figure 4.11，如果想了解链接（ld）的具体过程，可以看看本节参考《Unix环境高级编程编程》第7章 "UnIx进程的环境"，P127和P13，ELF: From The Programmer's Perspective，GNU-ld 连接脚本 Linker Scripts。

上面的操作对堆栈的操作比较少，下面我们用一个例子来演示栈在内存中的情况。

栈在内存中的组织

这一节主要介绍一个函数被调用时，参数是如何传递的，局部变量是如何存储的，它们对应的栈的位置和变化情况，从而加深对栈的理解。在操作时发现和参考资料的结果不太一样（参考资料中没有 `edi` 和 `esi` 相关信息，再第二部分的一个小程序里头也没有），可能是 `gcc` 版本的问题或者是它对不同源代码的处理不同。我的版本是 `4.1.2` （可以通过 `gcc --version` 查看）。

先来一段简单的程序，这个程序除了做一个加法操作外，还复制了一些字符串。

```
/* testshellcode.c */
#include <stdio.h>      /* printf */
#include <string.h>     /* memset, memcpy */

#define BUF_SIZE 8

#ifndef STR_SRC
# define STR_SRC "AAAAAAA"
#endif

int func(int a, int b, int c)
{
    int sum = 0;
    char buffer[BUF_SIZE];

    sum = a + b + c;

    memset(buffer, '\0', BUF_SIZE);
    memcpy(buffer, STR_SRC, sizeof(STR_SRC)-1);

    return sum;
}

int main()
{
    int sum;

    sum = func(1, 2, 3);

    printf("sum = %d\n", sum);

    return 0;
}
```

上面这个代码没有什么问题，编译执行一下：

```
$ make testshellcode
cc     testshellcode.c   -o testshellcode
$ ./testshellcode
sum = 6
```

下面调试一下，看看在调用 `func` 后的栈的内容。

```
$ gcc -g -o testshellcode testshellcode.c  #为了调试，需要在编译时加-g选项
$ gdb -q ./testshellcode   #启动gdb调试
...
(gdb) set logging on    #如果要记录调试过程中的信息，可以把日志记录功能打开
Copying output to gdb.txt.
(gdb) l main            #列出源代码
20
21              return sum;
22      }
23
24      int main()
25      {
26              int sum;
27
28              sum = func(1, 2, 3);
29
(gdb) break 28   #在调用func函数之前让程序停一下，以便记录当时的ebp(基指针)
Breakpoint 1 at 0x80483ac: file testshellcode.c, line 28.
(gdb) break func #设置断点在函数入口，以便逐步记录栈信息
Breakpoint 2 at 0x804835c: file testshellcode.c, line 13.
(gdb) disassemble main   #反编译main函数，以便记录调用func后的下一条指令地址
Dump of assembler code for function main:
0x0804839b <main+0>:    lea    0x4(%esp),%ecx
0x0804839f <main+4>:    and    $0xfffffff0,%esp
0x080483a2 <main+7>:    pushl  0xfffffffc(%ecx)
0x080483a5 <main+10>:   push   %ebp
0x080483a6 <main+11>:   mov    %esp,%ebp
0x080483a8 <main+13>:   push   %ecx
```

```
0x080483a9 <main+14>:   sub    $0x14,%esp
0x080483ac <main+17>:   push   $0x3
0x080483ae <main+19>:   push   $0x2
0x080483b0 <main+21>:   push   $0x1
0x080483b2 <main+23>:   call   0x8048354 <func>
0x080483b7 <main+28>:   add    $0xc,%esp
0x080483ba <main+31>:   mov    %eax,0xfffffff8(%ebp)
0x080483bd <main+34>:   sub    $0x8,%esp
0x080483c0 <main+37>:   pushl  0xfffffff8(%ebp)
0x080483c3 <main+40>:   push   $0x80484c0
0x080483c8 <main+45>:   call   0x80482a0 <printf@plt>
0x080483cd <main+50>:   add    $0x10,%esp
0x080483d0 <main+53>:   mov    $0x0,%eax
0x080483d5 <main+58>:   mov    0xfffffffc(%ebp),%ecx
0x080483d8 <main+61>:   leave
0x080483d9 <main+62>:   lea    0xfffffffc(%ecx),%esp
0x080483dc <main+65>:   ret
End of assembler dump.
(gdb) r        #运行程序
Starting program: /mnt/hda8/Temp/c/program/testshellcode

Breakpoint 1, main () at testshellcode.c:28
28              sum = func(1, 2, 3);
(gdb) print $ebp  #打印调用func函数之前的基地址，即Previous frame poi
nter。
$1 = (void *) 0xbf84fdd8
(gdb) n           #执行call指令并跳转到func函数的入口

Breakpoint 2, func (a=1, b=2, c=3) at testshellcode.c:13
13              int sum = 0;
(gdb) n
16              sum = a + b + c;
(gdb) x/11x $esp  #打印当前栈的内容，可以看出，地址从低到高，注意标记有蓝
色和红色的值
                 #它们分别是前一个栈基地址(ebp)和call调用之后的下一条指
令的指针(eip)
0xbf84fd94:     0x00000000      0x00000000      0x080482e0
0x00000000
0xbf84fda4:     0xb7f2bce0      0x00000000      0xbf84fdd8
0x080483b7
```

```
0xbf84fdb4:     0x00000001      0x00000002      0x00000003
(gdb) n       #执行sum = a + b + c，后，比较栈内容第一行，第4列，由0变
为6
18              memset(buffer, '\0', BUF_SIZE);
(gdb) x/11x $esp
0xbf84fd94:     0x00000000      0x00000000      0x080482e0
0x00000006
0xbf84fda4:     0xb7f2bce0      0x00000000      0xbf84fdd8
0x080483b7
0xbf84fdb4:     0x00000001      0x00000002      0x00000003
(gdb) n
19              memcpy(buffer, STR_SRC, sizeof(STR_SRC)-1);
(gdb) x/11x $esp #缓冲区初始化以后变成了0
0xbf84fd94:     0x00000000      0x00000000      0x00000000
0x00000006
0xbf84fda4:     0xb7f2bce0      0x00000000      0xbf84fdd8
0x080483b7
0xbf84fdb4:     0x00000001      0x00000002      0x00000003
(gdb) n
21              return sum;
(gdb) x/11x $esp #进行copy以后，这两列的值变了，大小刚好是7个字节，最后
一个字节为'\0'
0xbf84fd94:     0x00000000      0x41414141      0x00414141
0x00000006
0xbf84fda4:     0xb7f2bce0      0x00000000      0xbf84fdd8
0x080483b7
0xbf84fdb4:     0x00000001      0x00000002      0x00000003
(gdb) c
Continuing.
sum = 6

Program exited normally.
(gdb) quit
```

从上面的操作过程，我们可以得出大概的栈分布(`func` 函数结束之前)如下：

地址	值(hex)	符号或者寄存器	注释
低地址			栈顶方向
0xbf84fd98	0x41414141	buf[0]	可以看出little endian(小端，重要的数据在前面)
0xbf84fd9c	0x00414141	buf[1]	
0xbf84fda0	0x00000006	sum	可见这上面都是func函数里头的局部变量
0xbf84fda4	0xb7f2bce0	esi	源索引指针，可以通过产生中间代码查看，貌似没什么作用
0xbf84fda8	0x00000000	edi	目的索引指针
0xbf84fdac	0xbf84fdd8	ebp	调用func之前的栈的基地址，以便调用函数结束之后恢复
0xbf84fdb0	0x080483b7	eip	调用func之前的指令指针，以便调用函数结束之后继续执行
0xbf84fdb4	0x00000001	a	第一个参数
0xbf84fdb8	0x00000002	b	第二个参数
0xbf84fdbc	0x00000003	c	第三个参数，可见参数是从最后一个开始压栈的
高地址			栈底方向

先说明一下 `edi` 和 `esi` 的由来（在上面的调试过程中我们并没有看到），是通过产生中间汇编代码分析得出的。

```
$ gcc -S testshellcode.c
```

在产生的 `testShellcode.s` 代码里头的 `func` 部分看到 `push ebp` 之后就 `push` 了 `edi` 和 `esi` 。但是搜索了一下代码，发现就这个函数里头引用了这两个寄存器，所以保存它们没什么么用，删除以后编译产生目标代码后证明是没用的。

```
$ cat testshellcode.s
...
func:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %edi
        pushl   %esi
...
        popl    %esi
        popl    %edi
        popl    %ebp
...
```

下面就不管这两部分（`edi` 和 `esi`）了，主要来分析和函数相关的这几部分在栈内的分布：

- 函数局部变量，在靠近栈顶一端
- 调用函数之前的栈的基地址（`ebp`，`Previous Frame Pointer`），在中间靠近栈顶方向
- 调用函数指令的下一条指令地址 `` ` （ ``eip`）`，在中间靠近栈底的方向
- 函数参数，在靠近栈底的一端，最后一个参数最先入栈

到这里，函数调用时的相关内容在栈内的分布就比较清楚了，在具体分析缓冲区溢出问题之前，我们再来看一个和函数关系很大的问题，即函数返回值的存储问题：函数的返回值存放在寄存器 `eax` 中。

先来看这段代码：

```
/**
 * test_return.c -- the return of a function is stored in regist
er eax
 */

#include <stdio.h>

int func()
{
        __asm__ ("movl $1, %eax");
}

int main()
{
        printf("the return of func: %d\n", func());

        return 0;
}
```

编译运行后，可以看到返回值为 1，刚好是我们在 `func` 函数中 `mov` 到 `eax` 中的“立即数” 1，因此很容易理解返回值存储在 `eax` 中的事实，如果还有疑虑，可以再看看汇编代码。在函数返回之后， `eax` 中的值当作了 `printf` 的参数压入了栈中，而在源代码中我们正是把 `func` 的结果作为 `printf` 的第二个参数的。

```
$ make test_return
cc     test_return.c   -o test_return
$ ./test_return
the return of func: 1
$ gcc -S test_return.c
$ cat test_return.s
...
        call    func
        subl    $8, %esp
        pushl   %eax      #printf的第二个参数，把func的返回值压入了栈
底
        pushl   $.LC0     #printf的第一个参数the return of func: %
d\n
        call    printf
...
```

对于系统调用，返回值也存储在 `eax` 寄存器中。

缓冲区溢出

实例分析：字符串复制

先来看一段简短的代码。

```
/* testshellcode.c */
#include <stdio.h>      /* printf */
#include <string.h>     /* memset, memcpy */

#define BUF_SIZE 8

#ifdef STR1
# define STR_SRC "AAAAAAA\0\1\0\0\0"
#endif

#ifndef STR_SRC
# define STR_SRC "AAAAAAA"
#endif

int func(int a, int b, int c)
{
        int sum = 0;
        char buffer[BUF_SIZE];

        sum = a + b + c;

        memset(buffer, '\0', BUF_SIZE);
        memcpy(buffer, STR_SRC, sizeof(STR_SRC)-1);

        return sum;
}

int main()
{
        int sum;

        sum = func(1, 2, 3);

        printf("sum = %d\n", sum);

        return 0;
}
```

编译一下看看结果：

```
$ gcc -DSTR1 -o testshellcode testshellcode.c  #通过-D定义宏STR1，从而采用第一个STR_SRC的值
$ ./testshellcode
sum = 1
```

不知道你有没有发现异常呢？上面用红色标记的地方，本来 `sum` 为 `1+2+3` 即 6，但是实际返回的竟然是 1 。到底是什么原因呢？大家应该有所了解了，因为我们在复制字符串 `AAAAAAA\\0\\1\\0\\0\\0` 到 `buf` 的时候超出 `buf` 本来的大小。 `buf` 本来的大小是 `BUF_SIZE` ，8 个字节，而我们要复制的内容是 12 个字节，所以超出了四个字节。根据第一小节的分析，我们用栈的变化情况来表示一下这个复制过程（即执行 `memcpy` 的过程）。

```
memcpy(buffer, STR_SRC, sizeof(STR_SRC)-1);

（低地址）
复制之前     ====> 复制之后
0x00000000       0x41414141      #char buf[8]
0x00000000       0x00414141
0x00000006       0x00000001      #int sum
（高地址）
```

下面通过 `gdb` 调试来确认一下(只摘录了一些片断)。

```
$ gcc -DSTR1 -g -o testshellcode testshellcode.c
$ gdb -q ./testshellcode
...
(gdb) l
21
22              memset(buffer, '\0', BUF_SIZE);
23              memcpy(buffer, STR_SRC, sizeof(STR_SRC)-1);
24
25              return sum;
...
(gdb) break 23
Breakpoint 1 at 0x804837f: file testshellcode.c, line 23.
(gdb) break 25
Breakpoint 2 at 0x8048393: file testshellcode.c, line 25.
(gdb) r
Starting program: /mnt/hda8/Temp/c/program/testshellcode

Breakpoint 1, func (a=1, b=2, c=3) at testshellcode.c:23
23              memcpy(buffer, STR_SRC, sizeof(STR_SRC)-1);
(gdb) x/3x $esp+4
0xbfec6bd8:     0x00000000      0x00000000      0x00000006
(gdb) n

Breakpoint 2, func (a=1, b=2, c=3) at testshellcode.c:25
25              return sum;
(gdb) x/3x $esp+4
0xbfec6bd8:     0x41414141      0x00414141      0x00000001
```

可以看出，因为 C 语言没有对数组的边界进行限制。我们可以往数组中存入预定义长度的字符串，从而导致缓冲区溢出。

缓冲区溢出后果

溢出之后的问题是导致覆盖栈的其他内容，从而可能改变程序原来的行为。

如果这类问题被“黑客”利用那将产生非常可怕的后果，小则让非法用户获取了系统权限，把你的服务器当成“僵尸”，用来对其他机器进行攻击，严重的则可能被人删除数据（所以备份很重要）。即使不被黑客利用，这类问题如果放在医疗领域，那

将非常危险，可能那个被覆盖的数字刚好是用来控制治疗癌症的辐射量的，一旦出错，那可能导致置人死地，当然，如果在航天领域，那可能就是好多个 0 的 `money` 甚至航天员的损失，呵呵，“缓冲区溢出，后果很严重！”

缓冲区溢出应对策略

那这个怎么办呢？貌似Linux下缓冲区溢出攻击的原理及对策提到有一个 `libsafe` 库，可以至少用来检测程序中出现的类似超出数组边界的问题。对于上面那个具体问题，为了保护 `sum` 不被修改，有一个小技巧，可以让求和操作在字符串复制操作之后来做，以便求和操作把溢出的部分给重写。这个呆伙在下面一块看效果吧。继续看看缓冲区的溢出吧。

先来看看这个代码，还是 `testShellcode.c` 的改进。

```
/* testshellcode.c */
#include <stdio.h>      /* printf */
#include <string.h> /* memset, memcpy */
#define BUF_SIZE 8

#ifdef STR1
# define STR_SRC "AAAAAAAa\1\0\0\0"
#endif
#ifdef STR2
# define STR_SRC "AAAAAAAa\1\0\0\0BBBBBBBB"
#endif
#ifdef STR3
# define STR_SRC "AAAAAAAa\1\0\0\0BBBBBBBBCCCC"
#endif
#ifdef STR4
# define STR_SRC "AAAAAAAa\1\0\0\0BBBBBBBBCCCCDDDD"
#endif

#ifndef STR_SRC
# define STR_SRC "AAAAAAA"
#endif

int func(int a, int b, int c)
{
        int sum = 0;
```

```
        char buffer[BUF_SIZE] = "";

        memset(buffer, '\0', BUF_SIZE);
        memcpy(buffer, STR_SRC, sizeof(STR_SRC)-1);

        sum = a + b + c;     //把求和操作放在复制操作之后可以在一定情况
下“保护”求和结果

        return sum;
}

int main()
{
        int sum;

        sum = func(1, 2, 3);

        printf("sum = %d\n", sum);

        return 0;
}
```

看看运行情况：

```
$ gcc -D STR2 -o testshellcode testshellcode.c   #再多复制8个字节，
结果和STR1时一样
                        #原因是edi,esi这两个没什么用的，覆盖了也没关系
$ ./testshellcode       #看到没？这种情况下，让整数操作在字符串复制之后
做可以“保护‘整数结果
sum = 6
$ gcc -D STR3 -o testshellcode testshellcode.c  #再多复制4个字节，
现在就会把ebp给覆盖
                                                    #了，这样当main函数
再要用ebp访问数据
                                                    #时就会出现访问非法内
存而导致段错误。
$ ./testshellcode
Segmentation fault
```

如果感兴趣，自己还可以用gdb类似之前一样来查看复制字符串以后栈的变化情况。

如何保护 ebp 不被修改

下面来做一个比较有趣的事情：如何设法保护我们的 `ebp` 不被修改。

首先要明确 `ebp` 这个寄存器的作用和“行为”，它是栈基地址，并且发现在调用任何一个函数时，这个 `ebp` 总是在第一条指令被压入栈中，并在最后一条指令（ `ret` ）之前被弹出。类似这样：

```
func:                            #函数
       pushl %ebp                #第一条指令
       ...
       popl %ebp                 #倒数第二条指令
       ret
```

还记得之前（第一部分）提到的函数的返回值是存储在 `eax` 寄存器中的么？如果我们在一个函数中仅仅做放这两条指令：

```
popl %eax
pushl %eax
```

那不就刚好有：

```
func:                            #函数
       pushl %ebp                #第一条指令
       popl %eax                 #把刚压入栈中的ebp弹出存放到eax中
       pushl %eax                #又把ebp压入栈
       popl %ebp                 #倒数第二条指令
       ret
```

这样我们没有改变栈的状态，却获得了 `ebp` 的值，如果在调用任何一个函数之前，获取这个 `ebp` ，并且在任何一条字符串复制语句（可能导致缓冲区溢出的语句）之后重新设置一下 `ebp` 的值，那么就可以保护 `ebp` 啦。具体怎么实现呢？看这个代码。

```
/* testshellcode.c */
#include <stdio.h>      /* printf */
#include <string.h> /* memset, memcpy */
#define BUF_SIZE 8

#ifdef STR1
# define STR_SRC "AAAAAAAa\1\0\0\0"
#endif
#ifdef STR2
# define STR_SRC "AAAAAAAa\1\0\0\0BBBBBBBB"
#endif
#ifdef STR3
# define STR_SRC "AAAAAAAa\1\0\0\0BBBBBBBBCCCC"
#endif
#ifdef STR4
# define STR_SRC "AAAAAAAa\1\0\0\0BBBBBBBBCCCCDDDD"
#endif

#ifndef STR_SRC
# define STR_SRC "AAAAAAA"
#endif

unsigned long get_ebp()
{
        __asm__ ("popl %eax;"
                                "pushl %eax;");
}

int func(int a, int b, int c, unsigned long ebp)
{
        int sum = 0;
        char buffer[BUF_SIZE] = "";

        sum = a + b + c;
        memset(buffer, '\0', BUF_SIZE);
        memcpy(buffer, STR_SRC, sizeof(STR_SRC)-1);
        *(unsigned long *)(buffer+20) = ebp;
        return sum;
}
```

```
int main()
{
        int sum, ebp;

        ebp = get_ebp();
        sum = func(1, 2, 3, ebp);

        printf("sum = %d\n", sum);

        return 0;
}
```

这段代码和之前的代码的不同有：

- 给 `func` 函数增加了一个参数 `ebp` ，（其实可以用全局变量替代的）
- 利用了刚介绍的原理定义了一个函数 `get_ebp` 以便获取老的 `ebp`
- 在 `main` 函数中调用 `func` 之前调用了 `get_ebp` ，并把它作为 `func` 的最后一个参数
- 在 `func` 函数中调用 `memcpy` 函数（可能发生缓冲区溢出的地方）之后添加了一条恢复设置 `ebp` 的语句，这条语句先把 `buffer+20` 这个地址（存放 `ebp` 的地址，你可以类似第一部分提到的用 `gdb` 来查看）强制转换为指向一个 `unsigned long` 型的整数（4 个字节），然后把它指向的内容修改为老的 `ebp` 。

看看效果：

```
$ gcc -D STR3 -o testshellcode testshellcode.c
$ ./testshellcode         #现在没有段错误了吧，因为ebp得到了“保护”
sum = 6
```

如何保护 eip 不被修改？

如果我们复制更多的字节过去了，比如再多复制四个字节进去，那么 `eip` 就被覆盖了。

```
$ gcc -D STR4 -o testshellcode testshellcode.c
$ ./testshellcode
Segmentation fault
```

同样会出现段错误，因为下一条指令的位置都被改写了，`func` 返回后都不知道要访问哪个”非法“地址啦。呵呵，如果是一个合法地址呢？

如果在缓冲区溢出时，`eip` 被覆盖了，并且被修改为了一条合法地址，那么问题就非常”有趣“了。如果这个地址刚好是调用func的那个地址，那么整个程序就成了死循环，如果这个地址指向的位置刚好有一段关机代码，那么系统正在运行的所有服务都将被关掉，如果那个地方是一段更恶意的代码，那就？你可以尽情想像哦。如果是黑客故意利用这个，那么那些代码貌似就叫做shellcode了。

有没有保护 `eip` 的办法呢？呵呵，应该是有的吧。不知道 `gas` 有没有类似 `masm` 汇编器中 `offset` 的伪操作指令（查找了一下，貌似没有），如果有的话在函数调用之前设置一个标号，在后面某个位置获取，再加上一个可能的偏移（包括 `call` 指令的长度和一些 `push` 指令等），应该可以算出来，不过貌似比较麻烦（或许你灵感大作，找到好办法了！），这里直接通过 `gdb` 反汇编求得它相对 `main` 的偏移算出来得了。求出来以后用它来”保护“栈中的值。

看看这个代码：

```
/* testshellcode.c */
#include <stdio.h>      /* printf */
#include <string.h> /* memset, memcpy */
#define BUF_SIZE 8

#ifdef STR1
# define STR_SRC "AAAAAAAa\1\0\0\0"
#endif
#ifdef STR2
# define STR_SRC "AAAAAAAa\1\0\0\0BBBBBBBB"
#endif
#ifdef STR3
# define STR_SRC "AAAAAAAa\1\0\0\0BBBBBBBBCCCC"
#endif
#ifdef STR4
# define STR_SRC "AAAAAAAa\1\0\0\0BBBBBBBBCCCCDDDD"
```

```
#endif

#ifndef STR_SRC
# define STR_SRC "AAAAAAA"
#endif

int main();
#define OFFSET  40

unsigned long get_ebp()
{
        __asm__ ("popl %eax;"
                                "pushl %eax;");
}

int func(int a, int b, int c, unsigned long ebp)
{
        int sum = 0;
        char buffer[BUF_SIZE] = "";

        memset(buffer, '\0', BUF_SIZE);
        memcpy(buffer, STR_SRC, sizeof(STR_SRC)-1);

        sum = a + b + c;

        *(unsigned long *)(buffer+20) = ebp;
        *(unsigned long *)(buffer+24) = (unsigned long)main+OFFS
ET;
        return sum;
}

int main()
{
        int sum, ebp;

        ebp = get_ebp();
        sum = func(1, 2, 3, ebp);

        printf("sum = %d\n", sum);
```

```
        return 0;
}
```

看看效果：

```
$ gcc -D STR4 -o testshellcode testshellcode.c
$ ./testshellcode
sum = 6
```

这样， `EIP` 也得到了“保护”（这个方法很糟糕的，呵呵）。

类似地，如果再多复制一些内容呢？那么栈后面的内容都将被覆盖，即传递给 `func` 函数的参数都将被覆盖，因此上面的方法，包括所谓的对 `sum` 和 `ebp` 等值的保护都没有任何意义了（如果再对后面的参数进行进一步的保护呢？或许有点意义，呵呵）。在这里，之所以提出类似这样的保护方法，实际上只是为了讨论一些有趣的细节并加深对缓冲区溢出这一问题的理解（或许有一些实际的价值哦，算是抛砖引玉吧）。

缓冲区溢出检测

要确实解决这类问题，从主观上讲，还得程序员来做相关的工作，比如限制将要复制的字符串的长度，保证它不超过当初申请的缓冲区的大小。

例如，在上面的代码中，我们在 `memcpy` 之前，可以加入一个判断，并且可以对缓冲区溢出进行很好的检查。如果能够设计一些比较好的测试实例把这些判断覆盖到，那么相关的问题就可以得到比较不错的检查了。

```
/* testshellcode.c */
#include <stdio.h>      /* printf */
#include <string.h> /* memset, memcpy */
#include <stdlib.h>     /* exit */
#define BUF_SIZE 8

#ifdef STR4
# define STR_SRC "AAAAAAAa\1\0\0\0BBBBBBBBCCCCDDDD"
#endif

#ifndef STR_SRC
```

```
# define STR_SRC "AAAAAAA"
#endif

int func(int a, int b, int c)
{
        int sum = 0;
        char buffer[BUF_SIZE] = "";

        memset(buffer, '\0', BUF_SIZE);
        if ( sizeof(STR_SRC)-1 > BUF_SIZE ) {
                printf("buffer overflow!\n");
                exit(-1);
        }
        memcpy(buffer, STR_SRC, sizeof(STR_SRC)-1);

        sum = a + b + c;

        return sum;
}

int main()
{
        int sum;

        sum = func(1, 2, 3);

        printf("sum = %d\n", sum);

        return 0;
}
```

现在的效果如下：

```
$ gcc -DSTR4 -g -o testshellcode testshellcode.c
$ ./testshellcode      #如果存在溢出，那么就会得到阻止并退出，从而阻止可
能的破坏
buffer overflow!
$ gcc -g -o testshellcode testshellcode.c
$ ./testshellcode
sum = 6
```

当然，如果能够在 C 标准里头加入对数组操作的限制可能会更好，或者在编译器中扩展对可能引起缓冲区溢出的语法检查。

缓冲区注入实例

最后给出一个利用上述缓冲区溢出来进行缓冲区注入的例子。也就是通过往某个缓冲区注入一些代码，并把eip修改为这些代码的入口从而达到破坏目标程序行为的目的。

这个例子来自Linux 下缓冲区溢出攻击的原理及对策，这里主要利用上面介绍的知识对它进行了比较详细的分析。

准备：把 C 语言函数转换为字符串序列

首先回到第一部分，看看那个 `Shellcode.c` 程序。我们想获取它的汇编代码，并以十六进制字节的形式输出，以便把这些指令当字符串存放起来，从而作为缓冲区注入时的输入字符串。下面通过 `gdb` 获取这些内容。

```
$ gcc -g -o shellcode shellcode.c
$ gdb -q ./shellcode
(gdb) disassemble main
Dump of assembler code for function main:
...
0x08048331 <main+13>:   push   %ecx
0x08048332 <main+14>:   jmp    0x8048354 <forward>
0x08048334 <main+16>:   pop    %esi
0x08048335 <main+17>:   mov    $0x4,%eax
0x0804833a <main+22>:   mov    $0x2,%ebx
0x0804833f <main+27>:   mov    %esi,%ecx
```

```
0x08048341 <main+29>:   mov    $0xc,%edx
0x08048346 <main+34>:   int    $0x80
0x08048348 <main+36>:   mov    $0x1,%eax
0x0804834d <main+41>:   mov    $0x0,%ebx
0x08048352 <main+46>:   int    $0x80
0x08048354 <forward+0>: call   0x8048334 <main+16>
0x08048359 <forward+5>: dec    %eax
0x0804835a <forward+6>: gs
0x0804835b <forward+7>: insb   (%dx),%es:(%edi)
0x0804835c <forward+8>: insb   (%dx),%es:(%edi)
0x0804835d <forward+9>: outsl  %ds:(%esi),(%dx)
0x0804835e <forward+10>:        and    %dl,0x6f(%edi)
0x08048361 <forward+13>:        jb     0x80483cf <__libc_csu_ini
t+79>
0x08048363 <forward+15>:        or     %fs:(%eax),%al
...
End of assembler dump.
(gdb) set logging on   #开启日志功能，记录操作结果
Copying output to gdb.txt.
(gdb) x/52bx main+14  #以十六进制单字节（字符）方式打印出shellcode的核
心代码
0x8048332 <main+14>:    0xeb    0x20    0x5e    0xb8    0x04
0x00    0x00   0x00
0x804833a <main+22>:    0xbb    0x02    0x00    0x00    0x00
0x89    0xf1   0xba
0x8048342 <main+30>:    0x0c    0x00    0x00    0x00    0xcd
0x80    0xb8   0x01
0x804834a <main+38>:    0x00    0x00    0x00    0xbb    0x00
0x00    0x00   0x00
0x8048352 <main+46>:    0xcd    0x80    0xe8    0xdb    0xff
0xff    0xff   0x48
0x804835a <forward+6>:  0x65    0x6c    0x6c    0x6f    0x20
0x57    0x6f   0x72
0x8048362 <forward+14>: 0x6c    0x64    0x0a    0x00
(gdb) quit
$ cat gdb.txt | sed -e "s/^.*://g;s/\t/\\\/g;s/^/\"/g;s/\$/\"/g"
  #把日志里头的内容处理一下，得到这样一个字符串
"\0xeb\0x20\0x5e\0xb8\0x04\0x00\0x00\0x00"
"\0xbb\0x02\0x00\0x00\0x00\0x89\0xf1\0xba"
"\0x0c\0x00\0x00\0x00\0xcd\0x80\0xb8\0x01"
```

```
"\0x00\0x00\0x00\0xbb\0x00\0x00\0x00\0x00"
"\0xcd\0x80\0xe8\0xdb\0xff\0xff\0xff\0x48"
"\0x65\0x6c\0x6c\0x6f\0x20\0x57\0x6f\0x72"
"\0x6c\0x64\0x0a\0x00"
```

注入：在 C 语言中执行字符串化的代码

得到上面的字符串以后我们就可以设计一段下面的代码啦。

```
/* testshellcode.c */
char shellcode[]="\xeb\x20\x5e\xb8\x04\x00\x00\x00"
"\xbb\x02\x00\x00\x00\x89\xf1\xba"
"\x0c\x00\x00\x00\xcd\x80\xb8\x01"
"\x00\x00\x00\xbb\x00\x00\x00\x00"
"\xcd\x80\xe8\xdb\xff\xff\xff\x48"
"\x65\x6c\x6c\x6f\x20\x57\x6f\x72"
"\x6c\x64\x0a\x00";

void callshellcode(void)
{
   int *ret;
   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;
}

int main()
{
        callshellcode();

        return 0;
}
```

运行看看，

```
$ gcc -o testshellcode testshellcode.c
$ ./testshellcode
Hello World
```

竟然打印出了 `Hello World` ，实际上，如果只是为了让 `Shellcode` 执行，有更简单的办法，直接把 `Shellcode` 这个字符串入口强制转换为一个函数入口，并调用就可以，具体见这段代码。

```
char shellcode[]="\xeb\x20\x5e\xb8\x04\x00\x00\x00"
"\xbb\x02\x00\x00\x00\x89\xf1\xba"
"\x0c\x00\x00\x00\xcd\x80\xb8\x01"
"\x00\x00\x00\xbb\x00\x00\x00\x00"
"\xcd\x80\xe8\xdb\xff\xff\xff\x48"
"\x65\x6c\x6c\x6f\x20\x57\x6f\x72"
"\x6c\x64\x0a\x00";

typedef void (* func)();            //定义一个指向函数的指针func，而函数的返回值和参数均为void

int main()
{
        (* (func)shellcode)();

        return 0;
}
```

注入原理分析

这里不那样做，为什么也能够执行到 `Shellcode` 呢？仔细分析一下 `callShellcode` 里头的代码就可以得到原因了。

```
int *ret;
```

这里定义了一个指向整数的指针， `ret` 占用 4 个字节（可以用 `sizeof(int *)` 算出）。

```
ret = (int *)&ret + 2;
```

这里把 `ret` 修改为它本身所在的地址再加上两个单位。 首先需要求出 `ret` 本身所在的位置，因为 `ret` 是函数的一个局部变量，它在栈中偏栈顶的地方。 然后呢？再增加两个单位，这个单位是 `sizeof(int)` ，即 4 个字节。这样，新的 `ret` 就是 `ret` 所在的位置加上 8 个字节，即往栈底方向偏移 8 个字节的位置。对于我们之前分析的 `Shellcode` ，那里应该是 `edi` ，但实际上这里并不是 `edi` ，可能是 `gcc` 在编译程序时有不同的处理，这里实际上刚好是 `eip` ，即执行这条语句之后 `ret` 的值变成了 `eip` 所在的位置。

```
(*ret) = (int)shellcode;
```

由于之前 `ret` 已经被修改为了 `eip` 所在的位置，这样对 `(*ret)` 赋值就会修改 `eip` 的值，即下一条指令的地址，这里把 `eip` 修改为了 `Shellcode` 的入口。因此，当函数返回时直接去执行 `Shellcode` 里头的代码，并打印了 `Hello World` 。

用 `gdb` 调试一下看看相关变量的值的情况。这里主要关心 `ret` 本身。 `ret` 本身是一个地址，首先它所在的位置变成了 `EIP` 所在的位置（把它自己所在的位置加上 `2*4` 以后赋于自己），然后， `EIP` 又指向了 `Shellcode` 处的代码。

```
$ gcc -g -o testshellcode testshellcode.c
$ gdb -q ./testshellcode
(gdb) l
8       void callshellcode(void)
9       {
10         int *ret;
11         ret = (int *)&ret + 2;
12         (*ret) = (int)shellcode;
13      }
14
15      int main()
16      {
17              callshellcode();
(gdb) break 17
Breakpoint 1 at 0x804834d: file testshell.c, line 17.
(gdb) break 11
Breakpoint 2 at 0x804832a: file testshell.c, line 11.
(gdb) break 12
Breakpoint 3 at 0x8048333: file testshell.c, line 12.
```

```
(gdb) break 13
Breakpoint 4 at 0x804833d: file testshell.c, line 13.
(gdb) r
Starting program: /mnt/hda8/Temp/c/program/testshell

Breakpoint 1, main () at testshell.c:17
17              callshellcode();
(gdb) print $ebp       #打印ebp寄存器里的值
$1 = (void *) 0xbfcfd2c8
(gdb) disassemble main
...
0x0804834d <main+14>:   call   0x8048324 <callshellcode>
0x08048352 <main+19>:   mov    $0x0,%eax
...
(gdb) n

Breakpoint 2, callshellcode () at testshell.c:11
11          ret = (int *)&ret + 2;
(gdb) x/6x $esp
0xbfcfd2ac:     0x08048389      0xb7f4eff4      0xbfcfd36c
0xbfcfd2d8
0xbfcfd2bc:     0xbfcfd2c8      0x08048352
(gdb) print &ret #分别打印出ret所在的地址和ret的值，刚好在ebp之上，我们发现这里并没有
       #之前的testshellcode代码中的edi和esi，可能是gcc在汇编的时候有不同处理。
$2 = (int **) 0xbfcfd2b8
(gdb) print ret
$3 = (int *) 0xbfcfd2d8 #这里的ret是个随机值
(gdb) n

Breakpoint 3, callshellcode () at testshell.c:12
12          (*ret) = (int)shellcode;
(gdb) print ret   #执行完ret = (int *)&ret + 2;后，ret变成了自己地址加上2*4，
                  #刚好是eip所在的位置。
$5 = (int *) 0xbfcfd2c0
(gdb) x/6x $esp
0xbfcfd2ac:     0x08048389      0xb7f4eff4      0xbfcfd36c
0xbfcfd2c0
```

```
0xbfcfd2bc:     0xbfcfd2c8      0x08048352
(gdb) x/4x *ret  #此时*ret刚好为eip，0x8048352
0x8048352 <main+19>:    0x000000b8      0x8d5d5900      0x90c3fc
61      0x89559090
(gdb) n

Breakpoint 4, callshellcode () at testshell.c:13
13      }
(gdb) x/6x $esp #现在eip被修改为了shellcode的入口
0xbfcfd2ac:     0x08048389      0xb7f4eff4      0xbfcfd36c
0xbfcfd2c0
0xbfcfd2bc:     0xbfcfd2c8      0x8049560
(gdb) x/4x *ret  #现在修改了(*ret)的值，即修改了eip的值，使eip指向了sh
ellcode
0x8049560 <shellcode>:  0xb85e20eb      0x00000004      0x000002
bb      0xbaf18900
```

上面的过程很难弄，呵呵。主要是指针不大好理解，如果直接把它当地址绘出下面的图可能会容易理解一些。

callshellcode栈的初始分布：

```
ret=(int *)&ret+2=0xbfcfd2bc+2*4=0xbfcfd2c0
0xbfcfd2b8      ret(随机值)                    0xbfcfd2c0
0xbfcfd2bc      ebp(这里不关心)
0xbfcfd2c0      eip(0x08048352)         eip(0x8049560 )

(*ret) = (int)shellcode;即eip=0x8049560
```

总之，最后体现为函数调用的下一条指令指针（ `eip` ）被修改为一段注入代码的入口，从而使得函数返回时执行了注入代码。

缓冲区注入与防范

这个程序里头的注入代码和被注入程序竟然是一个程序，傻瓜才自己攻击自己（不过有些黑客有可能利用程序中一些空闲空间注入代码哦），真正的缓冲区注入程序是分开的，比如作为被注入程序的一个字符串参数。而在被注入程序中刚好没有做字符串长度的限制，从而让这段字符串中的一部分修改了 `eip` ，另外一部分作为

注入代码运行了，从而实现了注入的目的。不过这会涉及到一些技巧，即如何刚好用注入代码的入口地址来修改 `eip` （即新的 `eip` 能够指向注入代码）？如果 `eip` 的位置和缓冲区的位置之间的距离是确定，那么就比较好处理了，但从上面的两个例子中我们发现，有一个编译后有 `edi` 和 `esi` ，而另外一个则没有，另外，缓冲区的位置，以及被注入程序有多少个参数我们都无法预知，因此，如何计算 `eip` 所在的位置呢？这也会很难确定。还有，为了防止缓冲区溢出带来的注入问题，现在的操作系统采取了一些办法，比如让 `esp` 随机变化（比如和系统时钟关联起来），所以这些措施将导致注入更加困难。如果有兴趣，你可以接着看看最后的几篇参考资料并进行更深入的研究。

需要提到的是，因为很多程序可能使用 `strcpy` 来进行字符串的复制，在实际编写缓冲区注入代码时，会采取一定的办法（指令替换），把代码中可能包含的 `\0` 字节去掉，从而防止 `strcpy` 中断对注入代码的复制，进而可以复制完整的注入代码。具体的技巧可以参考 Linux下缓冲区溢出攻击的原理及对策，Shellcode技术杂谈，virus-writing-HOWTO。

后记

实际上缓冲区溢出应该是语法和逻辑方面的双重问题，由于语法上的不严格（对数组边界没有检查）导致逻辑上可能出现严重缺陷（程序执行行为被改变）。另外，这类问题是对程序运行过程中的程序映像的栈区进行注入。实际上除此之外，程序在安全方面还有很多类似的问题。比如，虽然程序映像的正文区受到系统保护（只读），但是如果内存（硬件本身，内存条）出现故障，在程序运行的过程中，程序映像的正文区的某些字节就可能被修改了，也可能发生非常严重的后果，因此程序运行过程的正文区检查等可能的手段需要被引入。

参考资料

- Playing with ptrace
    - how ptrace can be used to trace system calls and change system call arguments
    - setting breakpoints and injecting code into running programs
    - fix the problem of ORIG_EAX not defined
- 《缓冲区溢出攻击—— 检测、剖析与预防》第五章
- Linux下缓冲区溢出攻击的原理及对策
- Linux 汇编语言开发指南

- Shellcode 技术杂谈

进程的内存映像

前言

在阅读《UNIX 环境高级编程》的第 14 章时，看到一个“打印不同类型的数据所存放的位置”的例子，它非常清晰地从程序内部反应了“进程的内存映像”，通过结合它与《Gcc 编译的背后》和《缓冲区溢出与注入分析》的相关内容，可以更好地辅助理解相关的内容。

进程内存映像表

首先回顾一下《缓冲区溢出与注入》中提到的"进程内存映像表"，并把共享内存的大概位置加入该表：

地址	内核空间	描述
0xC0000000		
	(program flie) 程序名	execve 的第一个参数
	(environment) 环境变量	execve 的第三个参数，main 的第三个参数
	(arguments) 参数	execve 的第二个参数，main 的形参
	(stack) 栈	自动变量以及每次函数调用时所需保存的信息都
		存放在此，包括函数返回地址、调用者的
		环境信息等，函数的参数，局部变量都存放在此
	(shared memory) 共享内存	共享内存的大概位置
	...	
	...	
	(heap) 堆	主要在这里进行动态存储分配，比如 malloc，new 等。
	...	
	.bss (uninitilized data)	没有初始化的数据（全局变量哦）
	.data (initilized global data)	已经初始化的全局数据（全局变量）
	.text (Executable Instructions)	通常是可执行指令
0x08048000		
0x00000000		...

在程序内部打印内存分布信息

为了能够反应上述内存分布情况，这里在《UNIX 环境高级编程》的程序 14-11 的基础上，添加了一个已经初始化的全局变量（存放在已经初始化的数据段内），并打印了它以及 `main` 函数(处在代码正文部分)的位置。

```
/**
 * showmemory.c -- print the position of different types of data
 in a program in the memory
 */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

#define ARRAY_SIZE 4000
#define MALLOC_SIZE 100000
#define SHM_SIZE 100000
#define SHM_MODE (SHM_R | SHM_W)    /* user read/write */

int init_global_variable = 5;    /* initialized global variable
*/
char array[ARRAY_SIZE];        /* uninitialized data = bss */

int main(void)
{
    int shmid;
    char *ptr, *shmptr;

    printf("main: the address of the main function is %x\n", mai
n);
    printf("data: data segment is from %x\n", &init_global_varia
ble);
    printf("bss: array[] from %x to %x\n", &array[0], &array[ARR
AY_SIZE]);
    printf("stack: around %x\n", &shmid);

    /* shmid is a local variable, which is stored in the stack,
hence, you
     * can get the address of the stack via it*/

    if ( (ptr = malloc(MALLOC_SIZE)) == NULL) {
        printf("malloc error!\n");
        exit(-1);
```

```
    }
    printf("heap: malloced from %x to %x\n", ptr, ptr+MALLOC_SIZ
E);

    if ( (shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0)
{
        printf("shmget error!\n");
        exit(-1);
    }

    if ( (shmptr = shmat(shmid, 0, 0)) == (void *) -1) {
        printf("shmat error!\n");
        exit(-1);
    }
    printf("shared memory: attached from %x to %x\n", shmptr, sh
mptr+SHM_SIZE);

    if (shmctl(shmid, IPC_RMID, 0) < 0) {
        printf("shmctl error!\n");
        exit(-1);
    }

    exit(0);
}
```

该程序的运行结果如下：

```
$  make showmemory
cc     showmemory.c   -o showmemory
$ ./showmemory
main: the address of the main function is 804846c
data: data segment is from 80498e8
bss: array[] from 8049920 to 804a8c0
stack: around bfe3e224
heap: malloced from 804b008 to 80636a8
shared memory: attached from b7da7000 to b7dbf6a0
```

上述运行结果反应了几个重要部分数据的大概分布情况，比如 `data` 段（那个初始化过的全局变量就位于这里）、bss 段、stack、heap，以及 shared memory 和 main（代码段）的内存分布情况。

在程序内部获取完整内存分布信息

不过，这些结果还是没有精确和完整地反应所有相关信息，如果要想在程序内完整反应这些信息，结合《Gcc编译的背后》，就不难想到，我们还可以通过扩展一些已经链接到可执行文件中的外部符号来获取它们。这些外部符号全部定义在可执行文件的符号表中，可以通过 `nm/readelf -s/objdump -t` 等查看到，例如：

```
$ nm showmemory
080497e4 d _DYNAMIC
080498b0 d _GLOBAL_OFFSET_TABLE_
080486c8 R _IO_stdin_used
         w _Jv_RegisterClasses
080497d4 d __CTOR_END__
080497d0 d __CTOR_LIST__
080497dc d __DTOR_END__
080497d8 d __DTOR_LIST__
080487cc r __FRAME_END__
080497e0 d __JCR_END__
080497e0 d __JCR_LIST__
080498ec A __bss_start
080498dc D __data_start
08048680 t __do_global_ctors_aux
08048414 t __do_global_dtors_aux
080498e0 D __dso_handle
         w __gmon_start__
0804867a T __i686.get_pc_thunk.bx
080497d0 d __init_array_end
080497d0 d __init_array_start
08048610 T __libc_csu_fini
08048620 T __libc_csu_init
         U __libc_start_main@@GLIBC_2.0
080498ec A _edata
0804a8c0 A _end
080486a8 T _fini
```

```
080486c4 R _fp_hw
08048328 T _init
080483f0 T _start
08049920 B array
08049900 b completed.1
080498dc W data_start
         U exit@@GLIBC_2.0
08048444 t frame_dummy
080498e8 D init_global_variable
0804846c T main
         U malloc@@GLIBC_2.0
080498e4 d p.0
         U printf@@GLIBC_2.0
         U shmat@@GLIBC_2.0
         U shmctl@@GLIBC_2.2
         U shmget@@GLIBC_2.0
```

第三列的符号在我们的程序中被扩展以后就可以直接引用，这些符号基本上就已经完整地覆盖了相关的信息了，这样就可以得到一个更完整的程序，从而完全反应上面提到的内存分布表的信息。

```
/**
 * showmemory.c -- print the position of different types of data
 in a program in the memory
 */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

#define ARRAY_SIZE 4000
#define MALLOC_SIZE 100000
#define SHM_SIZE 100000
#define SHM_MODE (SHM_R | SHM_W)        /* user read/write */

                                        /* declare the address r
elative variables */
```

```
extern char _start, __data_start, __bss_start, etext, edata, end
;
extern char **environ;

char array[ARRAY_SIZE];         /* uninitialized data = bss */

int main(int argc, char *argv[])
{
    int shmid;
    char *ptr, *shmptr;

    printf("===== memory map =====\n");
    printf(".text:\t0x%x->0x%x (_start, code text)\n", &_start,
&etext);
    printf(".data:\t0x%x->0x%x (__data_start, initilized data)\n
", &__data_start, &edata);
    printf(".bss: \t0x%x->0x%x (__bss_start, uninitilized data)\
n", &__bss_start, &end);

    /* shmid is a local variable, which is stored in the stack,
hence, you
     * can get the address of the stack via it*/

    if ( (ptr = malloc(MALLOC_SIZE)) == NULL) {
        printf("malloc error!\n");
        exit(-1);
    }

    printf("heap: \t0x%x->0x%x (address of the malloc space)\n",
 ptr, ptr+MALLOC_SIZE);

    if ( (shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0)
{
        printf("shmget error!\n");
        exit(-1);
    }

    if ( (shmptr = shmat(shmid, 0, 0)) == (void *) -1) {
        printf("shmat error!\n");
        exit(-1);
```

```
    }
    printf("shm  :\t0x%x->0x%x (address of shared memory)\n", sh
mptr, shmptr+SHM_SIZE);

    if (shmctl(shmid, IPC_RMID, 0) < 0) {
        printf("shmctl error!\n");
        exit(-1);
    }

    printf("stack:\t <--0x%x--> (address of local variables)\n",
 &shmid);
    printf("arg:  \t0x%x (address of arguments)\n", argv);
    printf("env:  \t0x%x (address of environment variables)\n",
environ);

    exit(0);
}
```

运行结果：

```
$ make showmemory
$ ./showmemory
===== memory map =====
.text:    0x8048440->0x8048754 (_start, code text)
.data:    0x8049a3c->0x8049a48 (__data_start, initilized data)
.bss:     0x8049a48->0x804aa20 (__bss_start, uninitilized data)
heap:     0x804b008->0x80636a8 (address of the malloc space)
shm  :    0xb7db6000->0xb7dce6a0 (address of shared memory)
stack:     <--0xbff85b64--> (address of local variables)
arg:      0xbff85bf4 (address of arguments)
env:      0xbff85bfc (address of environment variables)
```

后记

上述程序完整地勾勒出了进程的内存分布的各个重要部分，这样就可以从程序内部获取跟程序相关的所有数据了，一个非常典型的例子是，在程序运行的过程中检查代码正文部分是否被恶意篡改。

如果想更深地理解相关内容，那么可以试着利用 `readelf` ， `objdump` 等来分析 ELF 可执行文件格式的结构，并利用 `gdb` 来了解程序运行过程中的内存变化情况。

参考资料

- Gcc 编译的背后（第二部分：汇编和链接）
- 缓冲区溢出与注入分析
- 《Unix 环境高级编程》第 14 章，程序 14-11

进程和进程的基本操作

前言

进程作为程序真正发挥作用时的“形态”，我们有必要对它的一些相关操作非常熟悉，这一节主要描述进程相关的概念和操作，将介绍包括程序、进程、作业等基本概念以及进程状态查询、进程通信等相关的操作。

什么是程序，什么又是进程

程序是指令的集合，而进程则是程序执行的基本单元。为了让程序完成它的工作，必须让程序运行起来成为进程，进而利用处理器资源、内存资源，进行各种 `I/O` 操作，从而完成某项特定工作。

从这个意思上说，程序是静态的，而进程则是动态的。

进程有区别于程序的地方还有：进程除了包含程序文件中的指令数据以外，还需要在内核中有一个数据结构用以存放特定进程的相关属性，以便内核更好地管理和调度进程，从而完成多进程协作的任务。因此，从这个意义上可以说“高于”程序，超出了程序指令本身。

如果进行过多进程程序的开发，又会发现，一个程序可能创建多个进程，通过多个进程的交互完成任务。在 Linux 下，多进程的创建通常是通过 `fork` 系统调用来实现。从这个意义上来说程序则”包含”了进程。

另外一个需要明确的是，程序可以由多种不同程序语言描述，包括 C 语言程序、汇编语言程序和最后编译产生的机器指令等。

下面简单讨论 Linux 下面如何通过 Shell 进行进程的相关操作。

进程的创建

通常在命令行键入某个程序文件名以后，一个进程就被创建了。例如，

让程序在后台运行

```
$ sleep 100 &
[1] 9298
```

查看进程 ID

用 `pidof` 可以查看指定程序名的进程ID：

```
$ pidof sleep
9298
```

查看进程的内存映像

```
$ cat /proc/9298/maps
08048000-0804b000 r-xp 00000000 08:01 977399     /bin/sleep
0804b000-0804c000 rw-p 00003000 08:01 977399     /bin/sleep
0804c000-0806d000 rw-p 0804c000 00:00 0          [heap]
b7c8b000-b7cca000 r--p 00000000 08:01 443354
...
bfbd8000-bfbed000 rw-p bfbd8000 00:00 0          [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]
```

程序被执行后，就被加载到内存中，成为了一个进程。上面显示了该进程的内存映像（虚拟内存），包括程序指令、数据，以及一些用于存放程序命令行参数、环境变量的栈空间，用于动态内存申请的堆空间都被分配好。

关于程序在命令行执行过程的细节，请参考《Linux 命令行下程序执行的一刹那》。

实际上，创建一个进程，也就是说让程序运行，还有其他的办法，比如，通过一些配置让系统启动时自动启动程序（具体参考 `man init`），或者是通过配置 `crond` （或者 `at`）让它定时启动程序。除此之外，还有一个方式，那就是编写 Shell 脚本，把程序写入一个脚本文件，当执行脚本文件时，文件中的程序将被执行而成为进程。这些方式的细节就不介绍，下面了解如何查看进程的属性。

需要补充一点的是：在命令行下执行程序，可以通过 `ulimit` 内置命令来设置进程可以利用的资源，比如进程可以打开的最大文件描述符个数，最大的栈空间，虚拟内存空间等。具体用法见 `help ulimit`。

查看进程的属性和状态

可以通过 `ps` 命令查看进程相关属性和状态，这些信息包括进程所属用户，进程对应的程序，进程对 `cpu` 和内存的使用情况等信息。熟悉如何查看它们有助于进行相关的统计分析等操作。

通过 ps 命令查看进程属性

查看系统当前所有进程的属性：

```
$ ps -ef
```

查看命令中包含某字符的程序对应的进程，进程 `ID` 是 1 。 `TTY` 为？表示和终端没有关联：

```
$ ps -C init
  PID TTY          TIME CMD
    1 ?        00:00:01 init
```

选择某个特定用户启动的进程：

```
$ ps -U falcon
```

按照指定格式输出指定内容，下面输出命令名和 `cpu` 使用率：

```
$ ps -e -o "%C %c"
```

打印 `cpu` 使用率最高的前 4 个程序：

```
$ ps -e -o "%C %c" | sort -u -k1 -r | head -5
 7.5 firefox-bin
 1.1 Xorg
 0.8 scim-panel-gtk
 0.2 scim-bridge
```

获取使用虚拟内存最大的 5 个进程：

```
$ ps -e -o "%z %c" | sort -n -k1 -r | head -5
349588 firefox-bin
 96612 xfce4-terminal
 88840 xfdesktop
 76332 gedit
 58920 scim-panel-gtk
```

通过 pstree 查看进程亲缘关系

系统所有进程之间都有“亲缘”关系，可以通过 `pstree` 查看这种关系：

```
$ pstree
```

上面会打印系统进程调用树，可以非常清楚地看到当前系统中所有活动进程之间的调用关系。

用top动态查看进程信息

```
$ top
```

该命令最大特点是可以动态地查看进程信息，当然，它还提供了一些其他的参数，比如 `-S` 可以按照累计执行时间的大小排序查看，也可以通过 `-u` 查看指定用户启动的进程等。

补充： `top` 命令支持交互式，比如它支持 `u` 命令显示用户的所有进程，支持通过 `k` 命令杀掉某个进程；如果使用 `-n 1` 选项可以启用批处理模式，具体用法为：

```
$ top -n 1 -b
```

确保特定程序只有一个副本在运行

下面来讨论一个有趣的问题：如何让一个程序在同一时间只有一个在运行。

这意味着当一个程序正在被执行时，它将不能再被启动。那该怎么做呢？

假如一份相同的程序被复制成了很多份，并且具有不同的文件名被放在不同的位置，这个将比较糟糕，所以考虑最简单的情况，那就是这份程序在整个系统上是唯一的，而且名字也是唯一的。这样的话，有哪些办法来回答上面的问题呢？

总的机理是：在程序开头检查自己有没有执行，如果执行了则停止否则继续执行后续代码。

策略则是多样的，由于前面的假设已经保证程序文件名和代码的唯一性，所以通过 `ps` 命令找出当前所有进程对应的程序名，逐个与自己的程序名比较，如果已经有，那么说明自己已经运行了。

```
ps -e -o "%c" | tr -d " " | grep -q ^init$   #查看当前程序是否执行
[ $? -eq 0 ] && exit   #如果在，那么退出, $?表示上一条指令是否执行成功
```

每次运行时先在指定位置检查是否存在一个保存自己进程 `ID` 的文件，如果不存在，那么继续执行，如果存在，那么查看该进程 `ID` 是否正在运行，如果在，那么退出，否则往该文件重新写入新的进程 `ID`，并继续。

```
pidfile=/tmp/$0".pid"
if [ -f $pidfile ]; then
       OLDPID=$(cat $pidfile)
    ps -e -o "%p" | tr -d " " | grep -q "^$OLDPID$"
    [ $? -eq 0 ] && exit
fi

echo $$ > $pidfile

#... 代码主体

#设置信号0的动作，当程序退出时触发该信号从而删除掉临时文件
trap "rm $pidfile"      0
```

更多实现策略自己尽情发挥吧！

调整进程的优先级

在保证每个进程都能够顺利执行外，为了让某些任务优先完成，那么系统在进行进程调度时就会采用一定的调度办法，比如常见的有按照优先级的时间片轮转的调度算法。这种情况下，可以通过 `renice` 调整正在运行的程序的优先级，例如：`

获取进程优先级

```
$ ps -e -o "%p %c %n" | grep xfs
 5089 xfs              0
```

调整进程的优先级

```
$ renice 1 -p 5089
renice: 5089: setpriority: Operation not permitted
$ sudo renice 1 -p 5089   #需要权限才行
[sudo] password for falcon:
5089: old priority 0, new priority 1
$ ps -e -o "%p %c %n" | grep xfs  #再看看，优先级已经被调整过来了
 5089 xfs              1
```

结束进程

既然可以通过命令行执行程序，创建进程，那么也有办法结束它。可以通过 `kill` 命令给用户自己启动的进程发送某个信号让进程终止，当然“万能”的 `root` 几乎可以 `kill` 所有进程（除了 `init` 之外）。例如，

结束进程

```
$ sleep 50 &   #启动一个进程
[1] 11347
$ kill 11347
```

`kill` 命令默认会发送终止信号（ `SIGTERM` ）给程序，让程序退出，但是 `kill` 还可以发送其他信号，这些信号的定义可以通过 `man 7 signal` 查看到，也可以通过 `kill -l` 列出来。

```
$ man 7 signal
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

暂停某个进程

例如，用 `kill` 命令发送 `SIGSTOP` 信号给某个程序，让它暂停，然后发送 `SIGCONT` 信号让它继续运行。

```
$ sleep 50 &
[1] 11441
$ jobs
[1]+  Running                 sleep 50 &
$ kill -s SIGSTOP 11441   #这个等同于我们对一个前台进程执行CTRL+Z操作
$ jobs
[1]+  Stopped                 sleep 50
$ kill -s SIGCONT 11441   #这个等同于之前我们使用bg %1操作让一个后台进
程运行起来
$ jobs
[1]+  Running                 sleep 50 &
$ kill %1                  #在当前会话(session)下，也可以通过作业号控
制进程
$ jobs
[1]+  Terminated              sleep 50
```

可见 `kill` 命令提供了非常好的功能，不过它只能根据进程的 `ID` 或者作业来控制进程，而 `pkill` 和 `killall` 提供了更多选择，它们扩展了通过程序名甚至是进程的用户名来控制进程的方法。更多用法请参考它们的手册。

查看进程退出状态

当程序退出后，如何判断这个程序是正常退出还是异常退出呢？还记得 Linux 下，那个经典 `hello world` 程序吗？在代码的最后总是有条 `return 0` 语句。这个 `return 0` 实际上是让程序员来检查进程是否正常退出的。如果进程返回了一个其他的数值，那么可以肯定地说这个进程异常退出了，因为它都没有执行到 `return 0` 这条语句就退出了。

那怎么检查进程退出的状态，即那个返回的数值呢？

在 `Shell` 中，可以检查这个特殊的变量 `$?`，它存放了上一条命令执行后的退出状态。

```
$ test1
bash: test1: command not found
$ echo $?
127
$ cat ./test.c | grep hello
$ echo $?
1
$ cat ./test.c | grep hi
    printf("hi, myself!\n");
$ echo $?
0
```

貌似返回 0 成为了一个潜规则，虽然没有标准明确规定，不过当程序正常返回时，总是可以从 `$?` 中检测到 0，但是异常时，总是检测到一个非 0 值。这就告诉我们在程序的最后最好是跟上一个 `exit 0` 以便任何人都可以通过检测 `$?` 确定程序是否正常结束。如果有一天，有人偶尔用到你的程序，试图检查它的退出状态，而你却在程序的末尾莫名地返回了一个 `-1` 或者 1，那么他将会很苦恼，会怀疑他自己编写的程序到底哪个地方出了问题，检查半天却不知所措，因为他太信任你了，竟然从头至尾都没有怀疑你的编程习惯可能会与众不同！

进程通信

为便于设计和实现，通常一个大型的任务都被划分成较小的模块。不同模块之间启动后成为进程，它们之间如何通信以便交互数据，协同工作呢？在《UNIX 环境高级编程》一书中提到很多方法，诸如管道（无名管道和有名管道）、信号（`signal`）、报文（`Message`）队列（消息队列）、共享内存（`mmap/munmap`）、信号量（`semaphore`，主要是同步用，进程之间，进程的不同线程之间）、套接口（`Socket`，支持不同机器之间的进程通信）等，而在 Shell 中，通常直接用到的就有管道和信号等。下面主要介绍管道和信号机制在 Shell 编程时的一些用法。

无名管道（pipe）

在 Linux 下，可以通过 `|` 连接两个程序，这样就可以用它来连接后一个程序的输入和前一个程序的输出，因此被形象地叫做个管道。在 C 语言中，创建无名管道非常简单方便，用 `pipe` 函数，传入一个具有两个元素的 `int` 型的数组就可以。

这个数组实际上保存的是两个文件描述符，父进程往第一个文件描述符里头写入东西后，子进程可以从第一个文件描述符中读出来。

如果用多了命令行，这个管子 `|` 应该会经常用。比如上面有个演示把 `ps` 命令的输出作为 `grep` 命令的输入：

```
$ ps -ef | grep init
```

也许会觉得这个“管子”好有魔法，竟然真地能够链接两个程序的输入和输出，它们到底是怎么实现的呢？实际上当输入这样一组命令时，当前 Shell 会进行适当的解析，把前面一个进程的输出关联到管道的输出文件描述符，把后面一个进程的输入关联到管道的输入文件描述符，这个关联过程通过输入输出重定向函数 `dup` （或者 `fcntl` ）来实现。

有名管道（named pipe）

有名管道实际上是一个文件（无名管道也像一个文件，虽然关系到两个文件描述符，不过只能一边读另外一边写），不过这个文件比较特别，操作时要满足先进先出，而且，如果试图读一个没有内容的有名管道，那么就会被阻塞，同样地，如果试图往一个有名管道里写东西，而当前没有程序试图读它，也会被阻塞。下面看看效果。

```
$ mkfifo fifo_test    #通过mkfifo命令创建一个有名管道
$ echo "fewfefe" > fifo_test
#试图往fifo_test文件中写入内容，但是被阻塞，要另开一个终端继续下面的操作
$ cat fifo_test        #另开一个终端，记得，另开一个。试图读出fifo_test的内容
fewfefe
```

这里的 `echo` 和 `cat` 是两个不同的程序，在这种情况下，通过 `echo` 和 `cat` 启动的两个进程之间并没有父子关系。不过它们依然可以通过有名管道通信。

这样一种通信方式非常适合某些特定情况：例如有这样一个架构，这个架构由两个应用程序构成，其中一个通过循环不断读取 `fifo_test` 中的内容，以便判断，它下一步要做什么。如果这个管道没有内容，那么它就会被阻塞在那里，而不会因死循环而耗费资源，另外一个则作为一个控制程序不断地往 `fifo_test` 中写入

一些控制信息，以便告诉之前的那个程序该做什么。下面写一个非常简单的例子。可以设计一些控制码，然后控制程序不断地往 `fifo_test` 里头写入，然后应用程序根据这些控制码完成不同的动作。当然，也可以往 `fifo_test` 传入除控制码外的其他数据。

- 应用程序的代码

```
$ cat app.sh
#!/bin/bash

FIFO=fifo_test
while :;
do
    CI=`cat $FIFO`  #CI --> Control Info
    case $CI in
        0) echo "The CONTROL number is ZERO, do something ..."
            ;;
        1) echo "The CONTROL number is ONE, do something ..."
            ;;
        *) echo "The CONTROL number not recognized, do something else..."
            ;;
    esac
done
```

- 控制程序的代码

```
$ cat control.sh
#!/bin/bash

FIFO=fifo_test
CI=$1

[ -z "$CI" ] && echo "the control info should not be empty
" && exit

echo $CI > $FIFO
```

- 一个程序通过管道控制另外一个程序的工作

```
$ chmod +x app.sh control.sh    #修改这两个程序的可执行权限，以
使用户可以执行它们
$ ./app.sh  #在一个终端启动这个应用程序，在通过./control.sh发送控
制码以后查看输出
The CONTROL number is ONE, do something ...    #发送1以后
The CONTROL number is ZERO, do something ...    #发送0以后
The CONTROL number not recognized, do something else...  #
发送一个未知的控制码以后
$ ./control.sh 1            #在另外一个终端，发送控制信息，控制
应用程序的工作
$ ./control.sh 0
$ ./control.sh 4343
```

这样一种应用架构非常适合本地的多程序任务设计，如果结合 `web cgi` ，那么也将适合远程控制的要求。引入 `web cgi` 的唯一改变是，要把控制程序 `./control.sh` 放到 `web` 的 `cgi` 目录下，并对它作一些修改，以使它符合 `CGI` 的规范，这些规范包括文档输出格式的表示（在文件开头需要输出 `content-tpye: text/html` 以及一个空白行）和输入参数的获取 `(web` 输入参数都存放在 `QUERY_STRING` 环境变量里头）。因此一个非常简单的 `CGI` 控制程序可以写成这样：

```
#!/bin/bash

FIFO=./fifo_test
CI=$QUERY_STRING

[ -z "$CI" ] && echo "the control info should not be empty" && e
xit

echo -e "content-type: text/html\n\n"
echo $CI > $FIFO
```

在实际使用时，请确保 `control.sh` 能够访问到 `fifo_test` 管道，并且有写权限，以便通过浏览器控制 `app.sh` ：

```
http://ipaddress\_or\_dns/cgi-bin/control.sh?0
```

问号 `?` 后面的内容即 `QUERY_STRING` ，类似之前的 `$1` 。

这样一种应用对于远程控制，特别是嵌入式系统的远程控制很有实际意义。在去年的暑期课程上，我们就通过这样一种方式来实现马达的远程控制。首先，实现了一个简单的应用程序以便控制马达的转动，包括转速，方向等的控制。为了实现远程控制，我们设计了一些控制码，以便控制马达转动相关的不同属性。

在 C 语言中，如果要使用有名管道，和 Shell 类似，只不过在读写数据时用 `read` ， `write` 调用，在创建 `fifo` 时用 `mkfifo` 函数调用。

信号（Signal）

信号是软件中断，Linux 用户可以通过 `kill` 命令给某个进程发送一个特定的信号，也可以通过键盘发送一些信号，比如 `CTRL+C` 可能触发 `SGIINT` 信号，而 `CTRL+\` 可能触发 `SGIQUIT` 信号等，除此之外，内核在某些情况下也会给进程发送信号，比如在访问内存越界时产生 `SGISEGV` 信号，当然，进程本身也可以通过 `kill` ， `raise` 等函数给自己发送信号。对于 Linux 下支持的信号类型，大家可以通过 `man 7 signal` 或者 `kill -l` 查看到相关列表和说明。

对于有些信号，进程会有默认的响应动作，而有些信号，进程可能直接会忽略，当然，用户还可以对某些信号设定专门的处理函数。在 Shell 中，可以通过 `trap` 命令（Shell 内置命令）来设定响应某个信号的动作（某个命令或者定义的某个函数），而在 C 语言中可以通过 `signal` 调用注册某个信号的处理函数。这里仅仅演示 `trap` 命令的用法。

```
$ function signal_handler { echo "hello, world."; } #定义signal_handler函数
$ trap signal_handler SIGINT  #执行该命令设定：收到SIGINT信号时打印hello, world
$ hello, world     #按下CTRL+C，可以看到屏幕上输出了hello, world字符串
```

类似地，如果设定信号 0 的响应动作，那么就可以用 `trap` 来模拟 C 语言程序中的 `atexit` 程序终止函数的登记，即通过 `trap signal_handler SIGQUIT` 设定的 `signal_handler` 函数将在程序退出时执行。信号 0 是一个特别的信号，在 `POSIX.1` 中把信号编号 0 定义为空信号，这常被用来确定一个特定进程是否仍旧存在。当一个程序退出时会触发该信号。

```
$ cat sigexit.sh
#!/bin/bash

function signal_handler {
    echo "hello, world"
}
trap signal_handler 0
$ chmod +x sigexit.sh
$ ./sigexit.sh    #实际Shell编程会用该方式在程序退出时来做一些清理临时文件的收尾工作
hello, world
```

作业和作业控制

当我们为完成一些复杂的任务而将多个命令通过 `|,\>,<, ;, (,)` 等组合在一起时，通常这个命令序列会启动多个进程，它们间通过管道等进行通信。而有时在执行一个任务的同时，还有其他的任务需要处理，那么就经常会在命令序列的最后加

上一个&，或者在执行命令后，按下 `CTRL+Z` 让前一个命令暂停。以便做其他的任务。等做完其他一些任务以后，再通过 `fg` 命令把后台任务切换到前台。这样一种控制过程通常被成为作业控制，而那些命令序列则被成为作业，这个作业可能涉及一个或者多个程序，一个或者多个进程。下面演示一下几个常用的作业控制操作。

创建后台进程，获取进程的作业号和进程号

```
$ sleep 50 &
[1] 11137
```

把作业调到前台并暂停

使用 Shell 内置命令 `fg` 把作业 1 调到前台运行，然后按下 `CTRL+Z` 让该进程暂停

```
$ fg %1
sleep 50
^Z
[1]+  Stopped                 sleep 50
```

查看当前作业情况

```
$ jobs            #查看当前作业情况，有一个作业停止
[1]+  Stopped                 sleep 50
$ sleep 100 &     #让另外一个作业在后台运行
[2] 11138
$ jobs            #查看当前作业情况，一个正在运行，一个停止
[1]+  Stopped                 sleep 50
[2]-  Running                 sleep 100 &
```

启动停止的进程并运行在后台

```
$ bg %1
[2]+ sleep 50 &
```

不过，要在命令行下使用作业控制，需要当前 Shell，内核终端驱动等对作业控制支持才行。

参考资料

- 《UNIX 环境高级编程》

打造史上最小可执行 ELF 文件（45 字节，可打印字符串）

前言

本文从减少可执行文件大小的角度分析了 `ELF` 文件，期间通过经典的 `Hello World` 实例逐步演示如何通过各种常用工具来分析 `ELF` 文件，并逐步精简代码。

为了能够尽量减少可执行文件的大小，我们必须了解可执行文件的格式，以及链接生成可执行文件时的后台细节（即最终到底有哪些内容被链接到了目标代码中）。通过选择合适的可执行文件格式并剔除对可执行文件的最终运行没有影响的内容，就可以实现目标代码的裁减。因此，通过探索减少可执行文件大小的方法，就相当于实践性地去探索了可执行文件的格式以及链接过程的细节。

当然，算法的优化和编程语言的选择可能对目标文件的大小有很大的影响，在本文最后我们会跟参考资料 [1] 的作者那样去探求一个打印 `Hello World` 的可执行文件能够小到什么样的地步。

可执行文件格式的选取

可执行文件格式的选择要满足的一个基本条件是：目标系统支持该可执行文件格式，资料 [2] 分析和比较了 `UNIX` 平台下的三种可执行文件格式，这三种格式实际上代表着可执行文件的一个发展过程：

- a.out 文件格式非常紧凑，只包含了程序运行所必须的信息（文本、数据、 `BSS` ），而且每个 `section` 的顺序是固定的。
- coff 文件格式虽然引入了一个节区表以支持更多节区信息，从而提高了可扩展性，但是这种文件格式的重定位在链接时就已经完成，因此不支持动态链接（不过扩展的 `coff` 支持）。
- elf 文件格式不仅动态链接，而且有很好的扩展性。它可以描述可重定位文件、可执行文件和可共享文件（动态链接库）三类文件。

下面来看看 `ELF` 文件的结构图：

```
文件头部(ELF Header)
程序头部表(Program Header Table)
节区1(Section1)
节区2(Section2)
节区3(Section3)
...
节区头部(Section Header Table)
```

无论是文件头部、程序头部表、节区头部表还是各个节区，都是通过特定的结构体 `(struct)`描述的，这些结构在 elf.h 文件中定义。文件头部用于描述整个文件的类型、大小、运行平台、程序入口、程序头部表和节区头部表等信息。例如，我们可以通过文件头部查看该 ELF` 文件的类型。

```
$ cat hello.c   #典型的hello, world程序
#include <stdio.h>

int main(void)
{
    printf("hello, world!\n");
    return 0;
}
$ gcc -c hello.c   #编译，产生可重定向的目标代码
$ readelf -h hello.o | grep Type   #通过readelf查看文件头部找出该类型
  Type:                              REL (Relocatable file)
$ gcc -o hello hello.o   #生成可执行文件
$ readelf -h hello | grep Type
  Type:                              EXEC (Executable file)
$ gcc -fpic -shared -Wl,-soname,libhello.so.0 -o libhello.so.0.0 hello.o  #生成共享库
$ readelf -h libhello.so.0.0 | grep Type
  Type:                              DYN (Shared object file)
```

那节区头部表（将简称节区表）和程序头部表有什么用呢？实际上前者只对可重定向文件有用，而后者只对可执行文件和可共享文件有用。

节区表是用来描述各节区的，包括各节区的名字、大小、类型、虚拟内存中的位置、相对文件头的位置等，这样所有节区都通过节区表给描述了，这样连接器就可以根据文件头部表和节区表的描述信息对各种输入的可重定位文件进行合适的链接，包括节区的合并与重组、符号的重定位（确认符号在虚拟内存中的地址）等，把各个可重定向输入文件链接成一个可执行文件（或者是可共享文件）。如果可执行文件中使用了动态连接库，那么将包含一些用于动态符号链接的节区。我们可以通过 `readelf -S` （或 `objdump -h` ）查看节区表信息。

```
$ readelf -S hello  #可执行文件、可共享库、可重定位文件默认都生成有节区表
...
Section Headers:
  [Nr] Name              Type            Addr     Off    Size
ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000
00      0   0  0
  [ 1] .interp           PROGBITS        08048114 000114 000013
00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048128 000128 000020
00   A  0   0  4
  [ 3] .hash             HASH            08048148 000148 000028
04   A  5   0  4
...
    [ 7] .gnu.version      VERSYM          0804822a 00022a 00000
a 02   A  5   0  2
...
  [11] .init             PROGBITS        08048274 000274 000030
00  AX  0   0  4
...
  [13] .text             PROGBITS        080482f0 0002f0 000148
00  AX  0   0 16
  [14] .fini             PROGBITS        08048438 000438 00001c
00  AX  0   0  4
...
```

三种类型文件的节区（各个常见节区的作用请参考资料[11])可能不一样，但是有几个节区，例如 `.text` ， `.data` ， `.bss` 是必须的，特别是 `.text` ，因为这个节区包含了代码。如果一个程序使用了动态链接库（引用了动态连接库中的某个函数），那么需要 `.interp` 节区以便告知系统使用什么动态连接器程序来进行动态符号链接，进行某些符号地址的重定位。通常， `.rel.text` 节区只有可重定向文件有，用于链接时对代码区进行重定向，而 `.hash` ， `.plt` ， `.got` 等节区则只有可执行文件（或可共享库）有，这些节区对程序的运行特别重要。还有一些节区，可能仅仅是用于注释，比如 `.comment` ，这些对程序的运行似乎没有影响，是可有可无的，不过有些节区虽然对程序的运行没有用处，但是却可以用来辅助对程序进行调试或者对程序运行效率有影响。

虽然三类文件都必须包含某些节区，但是节区表对可重定位文件来说才是必须的，而程序的执行却不需要节区表，只需要程序头部表以便知道如何加载和执行文件。不过如果需要对可执行文件或者动态连接库进行调试，那么节区表却是必要的，否则调试器将不知道如何工作。下面来介绍程序头部表，它可通过 `readelf -l`（或 `objdump -p`）查看。

```
$ readelf -l hello.o #对于可重定向文件，gcc没有产生程序头部，因为它对可
重定向文件没用

There are no program headers in this file.
$  readelf -l hello  #而可执行文件和可共享文件都有程序头部
...
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz
Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000e0 0x000e0
R E 0x4
  INTERP         0x000114 0x08048114 0x08048114 0x00013 0x00013
R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x00470 0x00470
R E 0x1000
  LOAD           0x000470 0x08049470 0x08049470 0x0010c 0x00110
RW  0x1000
  DYNAMIC        0x000484 0x08049484 0x08049484 0x000d0 0x000d0
RW  0x4
  NOTE           0x000128 0x08048128 0x08048128 0x00020 0x00020
R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000
RW  0x4

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .gnu.hash .dynsym .dynstr
.gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .text .
fini .rodata .eh_frame
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag
   06
$ readelf -l libhello.so.0.0  #节区和上面类似，这里省略
```

从上面可看出程序头部表描述了一些段（`Segment`），这些段对应着一个或者多个节区，上面的 `readelf -l` 很好地显示了各个段与节区的映射。这些段描述了段的名字、类型、大小、第一个字节在文件中的位置、将占用的虚拟内存大小、在虚拟内存中的位置等。这样系统程序解释器将知道如何把可执行文件加载到内存中以及进行动态链接等动作。

该可执行文件包含 7 个段，`PHDR` 指程序头部，`INTERP` 正好对应 `.interp` 节区，两个 `LOAD` 段包含程序的代码和数据部分，分别包含有 `.text` 和 `.data`，`.bss` 节区，`DYNAMIC` 段包含 `.daynamic`，这个节区可能包含动态连接库的搜索路径、可重定位表的地址等信息，它们用于动态连接器。`NOTE` 和 `GNU_STACK` 段貌似作用不大，只是保存了一些辅助信息。因此，对于一个不使用动态连接库的程序来说，可能只包含 `LOAD` 段，如果一个程序没有数据，那么只有一个 `LOAD` 段就可以了。

总结一下，Linux 虽然支持很多种可执行文件格式，但是目前 `ELF` 较通用，所以选择 `ELF` 作为我们的讨论对象。通过上面对 `ELF` 文件分析发现一个可执行的文件可能包含一些对它的运行没用的信息，比如节区表、一些用于调试、注释的节区。如果能够删除这些信息就可以减少可执行文件的大小，而且不会影响可执行文件的正常运行。

链接优化

从上面的讨论中已经接触了动态连接库。`ELF` 中引入动态连接库后极大地方便了公共函数的共享，节约了磁盘和内存空间，因为不再需要把那些公共函数的代码链接到可执行文件，这将减少了可执行文件的大小。

与此同时，静态链接可能会引入一些对代码的运行可能并非必须的内容。你可以从《GCC 编译的背后（第二部分：汇编和链接）》了解到 `GCC` 链接的细节。从那篇 Blog 中似乎可以得出这样的结论：仅仅从是否影响一个 C 语言程序运行的角度上说，`GCC` 默认链接到可执行文件的几个可重定位文件（`crt1.o`，`rti.o`，`crtbegin.o`，`crtend.o`，`crtn.o`）并不是必须的，不过值得注意的是，如果没有链接那些文件但在程序末尾使用了 `return` 语句，`main` 函数将无法返回，因此需要替换为 `_exit` 调用；另外，既然程序在进入 `main` 之前有一个入口，那么 `main` 入口就不是必须的。因此，如果不采用默认链接也可以减少可执行文件的大小。

可执行文件“减肥”实例（从6442到708字节）

这里主要是根据上面两点来介绍如何减少一个可执行文件的大小。以 `Hello World` 为例。

首先来看看默认编译产生的 `Hello World` 的可执行文件大小。

系统默认编译

代码同上，下面是一组演示，

```
$ uname -r    #先查看内核版本和gcc版本，以便和你的结果比较
2.6.22-14-generic
$ gcc --version
gcc (GCC) 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)
...
$ gcc -o hello hello.c    #默认编译
$ wc -c hello    #产生一个大小为6442字节的可执行文件
6442 hello
```

不采用默认编译

可以考虑编辑时就把 `return 0` 替换成 `_exit(0)` 并包含定义该函数的 `unistd.h` 头文件。下面是从《GCC 编译的背后（第二部分：汇编和链接）》总结出的 `Makefile` 文件。

```
#file: Makefile
#functin: for not linking a program as the gcc do by default
#author: falcon<zhangjinw@gmail.com>
#update: 2008-02-23

MAIN = hello
SOURCE =
OBJS = hello.o
TARGET = hello
CC = gcc-3.4 -m32
LD = ld -m elf_i386

CFLAGSs += -S
CFLAGSc += -c
LDFLAGS += -dynamic-linker /lib/ld-linux.so.2 -L /usr/lib/ -L /l
ib -lc
RM = rm -f
SEDc = sed -i -e '/\#include[ "<]*unistd.h[ ">]*/d;' \
    -i -e '1i \#include <unistd.h>' \
    -i -e 's/return 0;/_exit(0);/'
SEDs = sed -i -e 's/main/_start/g'

all: $(TARGET)

$(TARGET):
    @$(SEDc) $(MAIN).c
    @$(CC) $(CFLAGSs) $(MAIN).c
    @$(SEDs) $(MAIN).s
    @$(CC) $(CFLAGSc) $(MAIN).s $(SOURCE)
    @$(LD) $(LDFLAGS) -o $@ $(OBJS)
clean:
    @$(RM) $(MAIN).s $(OBJS) $(TARGET)
```

把上面的代码复制到一个Makefile文件中，并利用它来编译hello.c。

```
$ make   #编译
$ ./hello   #这个也是可以正常工作的
Hello World
$ wc -c hello   #但是大小减少了4382个字节，减少了将近 70%
2060 hello
$ echo "6442-2060" | bc
4382
$ echo "(6442-2060)/6442" | bc -l
.68022353306426575597
```

对于一个比较小的程序，能够减少将近 70% “没用的”代码。

删除对程序运行没有影响的节区

使用上述 `Makefile` 来编译程序，不链接那些对程序运行没有多大影响的文件，实际上也相当于删除了一些“没用”的节区，可以通过下列演示看出这个实质。

```
$ make clean
$ make
$ readelf -l hello | grep "0[0-9]\ \ "
   00
   01     .interp
   02     .interp .hash .dynsym .dynstr .gnu.version .gnu.versio
n_r .rel.plt .plt .text .rodata
   03     .dynamic .got.plt
   04     .dynamic
   05
$ make clean
$ gcc -o hello hello.c
$ readelf -l hello | grep "0[0-9]\ \ "
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .gnu.hash .dynsym .dynstr
.gnu.version .gnu.version_r
      .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag
   06
```

通过比较发现使用自定义的 `Makefile` 文件，少了这么多节区： `.bss .ctors .data .dtors .eh_frame .fini .gnu.hash .got .init .jcr .note.ABI-tag .rel.dyn` 。再看看还有哪些节区可以删除呢？通过之前的分析发现有些节区是必须的，那 `.hash?.gnu.version?` 呢，通过 `strip -R` （或 `objcop -R` ）删除这些节区试试。

```
$ wc -c hello   #查看大小，以便比较
2060
$ time ./hello    #我们比较一下一些节区对执行时间可能存在的影响
Hello World

real    0m0.001s
user    0m0.000s
sys     0m0.000s
$ strip -R .hash hello   #删除.hash节区
$ wc -c hello
1448 hello
$ echo "2060-1448" | bc   #减少了612字节
612
$ time ./hello           #发现执行时间长了一些（实际上也可能是进程调度的问题）
Hello World

real    0m0.006s
user    0m0.000s
sys     0m0.000s
$ strip -R .gnu.version hello   #删除.gnu.version还是可以工作
$ wc -c hello
1396 hello
$ echo "1448-1396" | bc      #又减少了52字节
52
$ time ./hello
Hello World

real    0m0.130s
user    0m0.004s
sys     0m0.000s
$ strip -R .gnu.version_r hello   #删除.gnu.version_r就不工作了
$ time ./hello
./hello: error while loading shared libraries: ./hello: unsupported version 0 of Verneed record
```

通过删除各个节区可以查看哪些节区对程序来说是必须的，不过有些节区虽然并不影响程序的运行却可能会影响程序的执行效率，这个可以上面的运行时间看出个大概。 通过删除两个“没用”的节区，我们又减少了 `52+612` ，即 664 字节。

删除可执行文件的节区表

用普通的工具没有办法删除节区表，但是参考资料[1]的作者已经写了这样一个工具。你可以从这里下载到那个工具，它是该作者写的一序列工具 `ELFkickers` 中的一个。

下载并编译（注：1.0 之前的版本才支持 32 位和正常编译，新版本在代码中明确限定了数据结构为 `Elf64`）：

```
$ git clone https://github.com/BR903/ELFkickers
$ cd ELFkickers/sstrip/
$ git checkout f0622afa    # 检出 1.0 版
$ make
```

然后复制到 `/usr/bin` 下，下面用它来删除节区表。

```
$ sstrip hello      #删除ELF可执行文件的节区表
$ ./hello           #还是可以正常运行，说明节区表对可执行文件的运行没有任
何影响
Hello World
$ wc -c hello       #大小只剩下708个字节了
708 hello
$ echo "1396-708" | bc  #又减少了688个字节。
688
```

通过删除节区表又把可执行文件减少了 688 字节。现在回头看看相对于 `gcc` 默认产生的可执行文件，通过删除一些节区和节区表到底减少了多少字节？减幅达到了多少？

```
$ echo "6442-708" | bc   #
5734
$ echo "(6442-708)/6442" | bc -l
.89009624340266997826
```

减少了 5734 多字节，减幅将近 `90%`，这说明：对于一个简短的 `hello.c` 程序而言，`gcc` 引入了将近 `90%` 的对程序运行没有影响的数据。虽然通过删除节区和节区表，使得最终的文件只有 708 字节，但是打印一个 `Hello World` 真的需

要这么多字节么？事实上未必，因为：

- 打印一段 `Hello World` 字符串，我们无须调用 `printf` ，也就无须包含动态连接库，因此 `.interp` ， `.dynamic` 等节区又可以去掉。为什么？我们可以直接使用系统调用 `(sys_write)来打印字符串。
- 另外，我们无须把 `Hello World` 字符串存放到可执行文件中？而是让用户把它当作参数输入。

下面，继续进行可执行文件的“减肥”。

用汇编语言来重写"Hello World"（76字节）

采用默认编译

先来看看 `gcc` 默认产生的汇编代码情况。通过 `gcc` 的 `-S` 选项可得到汇编代码。

```
$ cat hello.c  #这个是使用_exit和printf函数的版本
#include <stdio.h>      /* printf */
#include <unistd.h>     /* _exit */

int main()
{
    printf("Hello World\n");
    _exit(0);
}
$ gcc -S hello.c    #生成汇编
$ cat hello.s       #这里是汇编代码
    .file   "hello.c"
    .section        .rodata
.LC0:
    .string "Hello World"
    .text
.globl main
    .type   main, @function
main:
    leal    4(%esp), %ecx
    andl    $-16, %esp
    pushl   -4(%ecx)
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ecx
    subl    $4, %esp
    movl    $.LC0, (%esp)
    call    puts
    movl    $0, (%esp)
    call    _exit
    .size   main, .-main
    .ident  "GCC: (GNU) 4.1.3 20070929 (prerelease) (Ubuntu 4.1.
2-16ubuntu2)"
    .section        .note.GNU-stack,"",@progbits
$ gcc -o hello hello.s   #看看默认产生的代码大小
$ wc -c hello
6523 hello
```

删除掉汇编代码中无关紧要内容

现在对汇编代码 `hello.s` 进行简单的处理得到，

```
.LC0:
    .string "Hello World"
    .text
.globl main
    .type   main, @function
main:
    leal    4(%esp), %ecx
    andl    $-16, %esp
    pushl   -4(%ecx)
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ecx
    subl    $4, %esp
    movl    $.LC0, (%esp)
    call    puts
    movl    $0, (%esp)
    call    _exit
```

再编译看看，

```
$ gcc -o hello.o hello.s
$ wc -c hello
6443 hello
$ echo "6523-6443" | bc   #仅仅减少了80个字节
80
```

不默认编译并删除掉无关节区和节区表

如果不采用默认编译呢并且删除掉对程序运行没有影响的节区和节区表呢？

```
$ sed -i -e "s/main/_start/g" hello.s   #因为没有初始化，所以得直接进入代码，替换main为_start
$ as --32 -o  hello.o hello.s
$ ld -melf_i386 -o hello hello.o --dynamic-linker /lib/ld-linux.so.2 -L /usr/lib -lc
$ ./hello
hello world!
$ wc -c hello
1812 hello
$ echo "6443-1812" | bc -l   #和之前的实验类似，也减少了4k左右
4631
$ readelf -l hello | grep "\ [0-9][0-9]\ "
   00
   01     .interp
   02     .interp .hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.plt .plt .text
   03     .dynamic .got.plt
   04     .dynamic
$ strip -R .hash hello
$ strip -R .gnu.version hello
$ wc -c hello
1200 hello
$ sstrip hello
$ wc -c hello  #这个结果比之前的708（在删除所有垃圾信息以后）个字节少了708-676，即32个字节
676 hello
$ ./hello
Hello World
```

容易发现这 32 字节可能跟节区 `.rodata` 有关系，因为刚才在链接完以后查看节区信息时，并没有 `.rodata` 节区。

用系统调用取代库函数

前面提到，实际上还可以不用动态连接库中的 `printf` 函数，也不用直接调用 `_exit` ，而是在汇编里头使用系统调用，这样就可以去掉和动态连接库关联的内容。如果想了解如何在汇编中使用系统调用，请参考资料 [9]。使用系统调用重写以后得到如下代码，

```
.LC0:
    .string "Hello World\xa\x0"
    .text
.global _start
_start:
    xorl   %eax, %eax
    movb   $4, %al                  #eax = 4, sys_write(fd, addr
, len)
    xorl   %ebx, %ebx
    incl   %ebx                     #ebx = 1, standard output
    movl   $.LC0, %ecx              #ecx = $.LC0, the address of
 string
    xorl   %edx, %edx
    movb   $13, %dl                 #edx = 13, the length of .st
ring
    int    $0x80
    xorl   %eax, %eax
    movl   %eax, %ebx               #ebx = 0
    incl   %eax                     #eax = 1, sys_exit
    int    $0x80
```

现在编译就不再需要动态链接器 `ld-linux.so` 了，也不再需要链接任何库。

```
$ as --32 -o hello.o hello.s
$ ld -melf_i386 -o hello hello.o
$ readelf -l hello

Elf file type is EXEC (Executable file)
Entry point 0x8048062
There are 1 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz
Flg Align
  LOAD           0x000000 0x08048000 0x08048000 0x0007b 0x0007b
R E 0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .text
$ sstrip hello
$ ./hello          #完全可以正常工作
Hello World
$ wc -c hello
123 hello
$ echo "676-123" | bc    #相对于之前，已经只需要123个字节了，又减少了553
个字节
553
```

可以看到效果很明显，只剩下一个 `LOAD` 段，它对应 `.text` 节区。

把字符串作为参数输入

不过是否还有办法呢？把 `Hello World` 作为参数输入，而不是硬编码在文件中。所以如果处理参数的代码少于 `Hello World` 字符串的长度，那么就可以达到减少目标文件大小的目的。

先来看一个能够打印程序参数的汇编语言程序，它来自参考资料[9]。

```
.text
.globl _start

_start:
    popl    %ecx            # argc
vnext:
    popl    %ecx            # argv
    test    %ecx, %ecx      # 空指针表明结束
    jz      exit
    movl    %ecx, %ebx
    xorl    %edx, %edx
strlen:
    movb    (%ebx), %al
    inc     %edx
    inc     %ebx
    test    %al, %al
    jnz     strlen
    movb    $10, -1(%ebx)
    movl    $4, %eax        # 系统调用号(sys_write)
    movl    $1, %ebx        # 文件描述符(stdout)
    int     $0x80
    jmp     vnext
exit:
    movl    $1,%eax         # 系统调用号(sys_exit)
    xorl    %ebx, %ebx      # 退出代码
    int     $0x80
    ret
```

编译看看效果，

```
$ as --32 -o args.o args.s
$ ld -melf_i386 -o args args.o
$ ./args "Hello World"  #能够打印输入的字符串，不错
./args
Hello World
$ sstrip args
$ wc -c args           #处理以后只剩下130字节
130 args
```

可以看到，这个程序可以接收用户输入的参数并打印出来，不过得到的可执行文件为 130 字节，比之前的 123 个字节还多了 7 个字节，看看还有改进么？分析上面的代码后，发现，原来的代码有些地方可能进行优化，优化后得到如下代码。

```
.global _start
_start:
    popl %ecx        #弹出argc
vnext:
    popl %ecx        #弹出argv[0]的地址
    test %ecx, %ecx  #空指针表明结束
    jz exit
    movl %ecx, %ebx  #复制字符串地址到ebx寄存器
    xorl %edx, %edx  #把字符串长度清零
strlen:                          #求输入字符串的长度
    movb (%ebx), %al           #复制字符到al，以便判断是否为字符串结束符\
0
    inc %edx                   #edx存放每个当前字符串的长度
    inc %ebx                   #ebx存放每个当前字符的地址
    test %al, %al              #判断字符串是否结束，即是否遇到\0
    jnz strlen
    movb $10, -1(%ebx)         #在字符串末尾插入一个换行符\0xa
    xorl %eax, %eax
    movb $4, %al               #eax = 4, sys_write(fd, addr, len)
    xorl %ebx, %ebx
    incl %ebx                  #ebx = 1, standard output
    int $0x80
    jmp vnext
exit:
    xorl %eax, %eax
    movl %eax, %ebx                    #ebx = 0
    incl %eax                  #eax = 1, sys_exit
    int $0x80
```

再测试（记得先重新汇编、链接并删除没用的节区和节区表）。

```
$ wc -c hello
124 hello
```

现在只有 124 个字节，不过还是比 123 个字节多一个，还有什么优化的办法么？

先来看看目前 `hello` 的功能，感觉不太符合要求，因为只需要打印 `Hello World` ，所以不必处理所有的参数，仅仅需要接收并打印一个参数就可以。这样的话，把 `jmp vnext` （2 字节）这个循环去掉，然后在第一个 `pop %ecx` 语句之前加一个 `pop %ecx` （1 字节）语句就可以。

```
.global _start
_start:
    popl %ecx
    popl %ecx        #弹出argc[0]的地址
    popl %ecx        #弹出argv[1]的地址
    test %ecx, %ecx
    jz exit
    movl %ecx, %ebx
    xorl %edx, %edx
strlen:
    movb (%ebx), %al
    inc %edx
    inc %ebx
    test %al, %al
    jnz strlen
    movb $10, -1(%ebx)
    xorl %eax, %eax
    movb $4, %al
    xorl %ebx, %ebx
    incl %ebx
    int $0x80
exit:
    xorl %eax, %eax
    movl %eax, %ebx
    incl %eax
    int $0x80
```

现在刚好 123 字节，和原来那个代码大小一样，不过仔细分析，还是有减少代码的余地：因为在这个代码中，用了一段额外的代码计算字符串的长度，实际上如果仅仅需要打印 `Hello World` ，那么字符串的长度是固定的，即 12 。所以这段代码

可去掉，与此同时测试字符串是否为空也就没有必要（不过可能影响代码健壮性！），当然，为了能够在打印字符串后就换行，在串的末尾需要加一个回车（ `$10` ）并且设置字符串的长度为 `12+1` ，即 13，

```
.global _start
_start:
    popl %ecx
    popl %ecx
    popl %ecx
    movb $10,12(%ecx) #在Hello World的结尾加一个换行符
    xorl %edx, %edx
    movb $13, %dl
    xorl %eax, %eax
    movb $4, %al
    xorl %ebx, %ebx
    incl %ebx
    int $0x80
    xorl %eax, %eax
    movl %eax, %ebx
    incl %eax
    int $0x80
```

再看看效果，

```
$ wc -c hello
111 hello
```

寄存器赋值重用

现在只剩下 111 字节，比刚才少了 12 字节。貌似到了极限？还有措施么？

还有，仔细分析发现：系统调用 `sys_exit` 和 `sys_write` 都用到了 `eax` 和 `ebx` 寄存器，它们之间刚好有那么一点巧合：

- sys_exit 调用时， `eax` 需要设置为 1， `ebx` 需要设置为 0 。
- sys_write 调用时， `ebx` 刚好是 1 。

因此，如果在 `sys_exit` 调用之前，先把 `ebx` 复制到 `eax` 中，再对 `ebx` 减一，则可减少两个字节。

不过，因为标准输入、标准输出和标准错误都指向终端，如果往标准输入写入一些东西，它还是会输出到标准输出上，所以在上述代码中如果在 `sys_write` 之前 `ebx` 设置为 0，那么也可正常往屏幕上打印 `Hello World` ，这样的话， `sys_exit` 调用前就没必要修改 `ebx` ，而仅需把 `eax` 设置为 1，这样就可减少 3 个字节。

```
.global _start
_start:
    popl %ecx
    popl %ecx
    popl %ecx
    movb $10,12(%ecx)
    xorl %edx, %edx
    movb $13, %dl
    xorl %eax, %eax
    movb $4, %al
    xorl %ebx, %ebx
    int $0x80
    xorl %eax, %eax
    incl %eax
    int $0x80
```

看看效果，

```
$ wc -c hello
108 hello
```

现在看一下纯粹的指令还有多少？

```
$ readelf -h hello | grep Size
  Size of this header:                52 (bytes)
  Size of program headers:            32 (bytes)
  Size of section headers:            0 (bytes)
$  echo "108-52-32" | bc
24
```

通过文件名传递参数

对于标准的 `main` 函数的两个参数，文件名实际上作为第二个参数（数组）的第一个元素传入，如果仅仅是为了打印一个字符串，那么可以打印文件名本身。例如，要打印 `Hello World` ，可以把文件名命名为 `Hello World` 即可。

这样地话，代码中就可以删除掉一条 `popl` 指令，减少 1 个字节，变成 107 个字节。

```
.global _start
_start:
    popl %ecx
    popl %ecx
    movb $10,12(%ecx)
    xorl %edx, %edx
    movb $13, %dl
    xorl %eax, %eax
    movb $4, %al
    xorl %ebx, %ebx
    int $0x80
    xorl %eax, %eax
    incl %eax
    int $0x80
```

看看效果，

```
$ as --32 -o hello.o hello.s
$ ld -melf_i386 -o hello hello.o
$ sstrip hello
$ wc -c hello
107
$ mv hello "Hello World"
$ export PATH=./:$PATH
$ Hello\ World
Hello World
```

删除非必要指令

在测试中发现，`edx`，`eax`，`ebx` 的高位即使不初始化，也常为 0，如果不考虑健壮性（仅这里实验用，实际使用中必须考虑健壮性），几条 `xorl` 指令可以移除掉。

另外，如果只是为了演示打印字符串，完全可以不用打印换行符，这样下来，代码可以综合优化成如下几条指令：

```
.global _start
_start:
    popl %ecx    # argc
    popl %ecx    # argv[0]
    movb $5, %dl    # 设置字符串长度
    movb $4, %al    # eax = 4, 设置系统调用号, sys_write(fd, addr, len) : ebx, ecx, edx
    int $0x80
    movb $1, %al
    int $0x80
```

看看效果：

```
$ as --32 -o hello.o hello.s
$ ld -melf_i386 -o hello hello.o
$ sstrip hello
$ wc -c hello
96
```

合并代码段、程序头和文件头（52字节）

把代码段移入文件头

纯粹的指令只有 `96-84=12` 个字节了，还有办法再减少目标文件的大小么？如果看了参考资料 [1]，看样子你又要蠢蠢欲动了：这 12 个字节是否可以插入到文件头部或程序头部？如果可以那是否意味着还可减少可执行文件的大小呢？现在来比较一下这三部分的十六进制内容。

```
$ hexdump -C hello -n 52     #文件头(52bytes)
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.EL
F............|
00000010  02 00 03 00 01 00 00 00  54 80 04 08 34 00 00 00  |...
.....T...4...|
00000020  00 00 00 00 00 00 00 00  34 00 20 00 01 00 00 00  |...
.....4. .....|
00000030  00 00 00 00                                       |...
.|
00000034
$ hexdump -C hello -s 52 -n 32    #程序头(32bytes)
00000034  01 00 00 00 00 00 00 00  00 80 04 08 00 80 04 08  |...
.............|
00000044  6c 00 00 00 6c 00 00 00  05 00 00 00 00 10 00 00  |l..
.l...........|
00000054
$ hexdump -C hello -s 84          #实际代码部分(12bytes)
00000054  59 59 b2 05 b0 04 cd 80  b0 01 cd 80              |YY.
.........|
00000060
```

从上面结果发现 `ELF` 文件头部和程序头部还有好些空洞（0），是否可以把指令字节分散放入到那些空洞里或者是直接覆盖掉那些系统并不关心的内容？抑或是把代码压缩以后放入可执行文件中，并在其中实现一个解压缩算法？还可以是通过一些代码覆盖率测试工具（ `gcov` ， `prof` ）对你的代码进行优化？

在继续介绍之前，先来看一个 `dd` 工具，可以用来直接“编辑” `ELF` 文件，例如，

直接往指定位置写入 `0xff` ：

```
$ hexdump -C hello -n 16    # 写入前，elf文件前16个字节
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.EL
F............|
00000010
$ echo -ne "\xff" | dd of=hello bs=1 count=1 seek=15 conv=notrun
c    # 把最后一个字节0覆盖掉
1+0 records in
1+0 records out
1 byte (1 B) copied, 3.7349e-05 s, 26.8 kB/s
$ hexdump -C hello -n 16    # 写入后果然被覆盖
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 ff  |.EL
F............|
00000010
```

- `seek=15` 表示指定写入位置为第 15 个（从第 0 个开始）
- `conv=notrunc` 选项表示要保留写入位置之后的内容，默认情况下会截断。
- `bs=1` 表示一次读/写 1 个
- `count=1` 表示总共写 1 次

覆盖多个连续的值：

把第 12，13，14，15 连续 4 个字节全部赋值为 `0xff` 。

```
$ echo -ne "\xff\xff\xff\xff" | dd of=hello bs=1 count=4 seek=12
 conv=notrunc
$ hexdump -C hello -n 16
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 ff ff ff ff  |.EL
F............|
00000010
```

下面，通过往文件头指定位置写入 `0xff` 确认哪些部分对于可执行文件的执行是否有影响？这里是逐步测试后发现依然能够执行的情况：

```
$ hexdump -C hello
00000000  7f 45 4c 46 ff ff ff ff  ff ff ff ff ff ff ff ff  |.EL
F............|
00000010  02 00 03 00 ff ff ff ff  54 80 04 08 34 00 00 00  |...
.....T...4...|
00000020  ff ff ff ff ff ff ff ff  34 00 20 00 01 00 ff ff  |...
.....4. .....|
00000030  ff ff ff ff 01 00 00 00  00 00 00 00 00 80 04 08  |...
.............|
00000040  00 80 04 08 60 00 00 00  60 00 00 00 05 00 00 00  |...
.`...`.......|
00000050  00 10 00 00 59 59 b2 05  b0 04 cd 80 b0 01 cd 80  |...
.YY..........|
00000060
```

可以发现，文件头部分，有 30 个字节即使被篡改后，该可执行文件依然可以正常执行。这意味着，这 30 字节是可以写入其他代码指令字节的。而我们的实际代码指令只剩下 12 个，完全可以直接移到前 12 个 `0xff` 的位置，即从第 4 个到第 15 个。

而代码部分的起始位置，通过 `readelf -h` 命令可以看到：

```
$ readelf -h hello | grep "Entry"
  Entry point address:               0x8048054
```

上面地址的最后两位 `0x54=84` 就是代码在文件中的偏移，也就是刚好从程序头之后开始的，也就是用文件头（52）+程序头（32）个字节开始的 12 字节覆盖到第 4 个字节开始的 12 字节内容即可。

上面的 `dd` 命令从 `echo` 命令获得输入，下面需要通过可执行文件本身获得输入，先把代码部分移过去：

```
$ dd if=hello of=hello bs=1 skip=84 count=12 seek=4 conv=notrunc
12+0 records in
12+0 records out
12 bytes (12 B) copied, 4.9552e-05 s, 242 kB/s
$ hexdump -C hello
00000000  7f 45 4c 46 59 59 b2 05  b0 04 cd 80 b0 01 cd 80  |.EL
FYY..........|
00000010  02 00 03 00 01 00 00 00  54 80 04 08 34 00 00 00  |...
.....T...4...|
00000020  00 00 00 00 00 00 00 00  34 00 20 00 01 00 00 00  |...
.....4. .....|
00000030  00 00 00 00 01 00 00 00  00 00 00 00 00 80 04 08  |...
.............|
00000040  00 80 04 08 60 00 00 00  60 00 00 00 05 00 00 00  |...
.`...`.......|
00000050  00 10 00 00 59 59 b2 05  b0 04 cd 80 b0 01 cd 80  |...
.YY..........|
00000060
```

接着把代码部分截掉：

```
$ dd if=hello of=hello bs=1 count=1 skip=84 seek=84
0+0 records in
0+0 records out
0 bytes (0 B) copied, 1.702e-05 s, 0.0 kB/s
$ hexdump -C hello
00000000  7f 45 4c 46 59 59 b2 05  b0 04 cd 80 b0 01 cd 80  |.EL
FYY..........|
00000010  02 00 03 00 01 00 00 00  54 80 04 08 34 00 00 00  |...
.....T...4...|
00000020  00 00 00 00 00 00 00 00  34 00 20 00 01 00 00 00  |...
.....4. .....|
00000030  00 00 00 00 01 00 00 00  00 00 00 00 00 80 04 08  |...
.............|
00000040  00 80 04 08 60 00 00 00  60 00 00 00 05 00 00 00  |...
.`...`.......|
00000050  00 10 00 00                                       |...
.|
00000054
```

这个时候还不能执行，因为代码在文件中的位置被移动了，相应地，文件头中的 `Entry point address` ，即文件入口地址也需要被修改为 `0x8048004` 。

即需要把 `0x54` 所在的第 24 个字节修改为 `0x04` ：

```
$ echo -ne "\x04" | dd of=hello bs=1 count=1 seek=24 conv=notrun
c
1+0 records in
1+0 records out
1 byte (1 B) copied, 3.7044e-05 s, 27.0 kB/s
$ hexdump -C hello
00000000  7f 45 4c 46 59 59 b2 05  b0 04 cd 80 b0 01 cd 80  |.EL
FYY..........|
00000010  02 00 03 00 01 00 00 00  04 80 04 08 34 00 00 00  |...
.........4...|
00000020  84 00 00 00 00 00 00 00  34 00 20 00 01 00 28 00  |...
.....4. ...(.|
00000030  05 00 02 00 01 00 00 00  00 00 00 00 00 80 04 08  |...
.............|
00000040  00 80 04 08 60 00 00 00  60 00 00 00 05 00 00 00  |...
.`...`.......|
00000050  00 10 00 00
```

修改后就可以执行了。

把程序头移入文件头

程序头部分经过测试发现基本上都不能修改并且需要是连续的，程序头有 32 个字节，而文件头中连续的 `0xff` 可以被篡改的只有从第 46 个开始的 6 个了，另外，程序头刚好是 `01 00` 开头，而第 44，45 个刚好为 `01 00` ，这样地话，这两个字节文件头可以跟程序头共享，这样地话，程序头就可以往文件头里头移动 8 个字节了。

```
$ dd if=hello of=hello bs=1 skip=52 seek=44 count=32 conv=notrun
c
```

再把最后 8 个没用的字节删除掉，保留 `84-8=76` 个字节：

```
$ dd if=hello of=hello bs=1 skip=76 seek=76
$ hexdump -C hello
00000000  7f 45 4c 46 59 59 b2 05  b0 04 cd 80 b0 01 cd 80  |.EL
FYY..........|
00000010  02 00 03 00 01 00 00 00  04 80 04 08 34 00 00 00  |...
.........4...|
00000020  84 00 00 00 00 00 00 00  34 00 20 00 01 00 00 00  |...
.....4. .....|
00000030  00 00 00 00 00 80 04 08  00 80 04 08 60 00 00 00  |...
.........`...|
00000040  60 00 00 00 05 00 00 00  00 10 00 00              |`..
.........|
0000004c
```

另外，还需要把文件头中程序头的位置信息改为 44，即第 28 个字节，原来是 `0x34`，即 52 的位置。

```
$ echo "obase=16;ibase=10;44" | bc    # 先把44转换是16进制的0x2C
2C
$ echo -ne "\x2C" | dd of=hello bs=1 count=1 seek=28 conv=notrun
c    # 修改文件头
1+0 records in
1+0 records out
1 byte (1 B) copied, 3.871e-05 s, 25.8 kB/s
$ hexdump -C hello
00000000  7f 45 4c 46 59 59 b2 05  b0 04 cd 80 b0 01 cd 80  |.EL
FYY..........|
00000010  02 00 03 00 01 00 00 00  04 80 04 08 2c 00 00 00  |...
.........,...|
00000020  84 00 00 00 00 00 00 00  34 00 20 00 01 00 00 00  |...
.....4. .....|
00000030  00 00 00 00 00 80 04 08  00 80 04 08 60 00 00 00  |...
.........`...|
00000040  60 00 00 00 05 00 00 00  00 10 00 00              |`..
.........|
0000004c
```

修改后即可执行了，目前只剩下 76 个字节：

```
$ wc -c hello
76
```

在非连续的空间插入代码

另外，还有 12 个字节可以放代码，见 `0xff` 的地方：

```
$ hexdump -C hello
00000000  7f 45 4c 46 59 59 b2 05  b0 04 cd 80 b0 01 cd 80  |.EL
FYY..........|
00000010  02 00 03 00 ff ff ff ff  04 80 04 08 2c 00 00 00  |...
.........,...|
00000020  ff ff ff ff ff ff ff ff  34 00 20 00 01 00 00 00  |...
.....4. .....|
00000030  00 00 00 00 00 80 04 08  00 80 04 08 60 00 00 00  |...
.........`...|
00000040  60 00 00 00 05 00 00 00  00 10 00 00              |`..
.........|
0000004c
```

不过因为空间不是连续的，需要用到跳转指令作为跳板利用不同的空间。

例如，如果要利用后面的 `0xff` 的空间，可以把第 14，15 位置的 `cd 80` 指令替换为一条跳转指令，比如跳转到第 20 个字节的位置，从跳转指令之后的 16 到 20 刚好 4 个字节。

然后可以参考 X86 指令编码表（也可以写成汇编生成可执行文件后用 `hexdump` 查看），可以把 `jmp` 指令编码为： `0xeb 0x04` 。

```
$ echo -ne "\xeb\x04" | dd of=hello bs=1 count=2 seek=14 conv=no
trunc
```

然后把原来位置的 `cd 80` 移动到第 20 个字节开始的位置：

```
$ echo -ne "\xcd\x80" | dd of=hello bs=1 count=2 seek=20 conv=no
trunc
```

依然可以执行，类似地可以利用更多非连续的空间。

把程序头完全合入文件头

在阅读参考资料 [1]后，发现有更多深层次的探讨，通过分析 Linux 系统对 `ELF` 文件头部和程序头部的解析，可以更进一步合并程序头和文件头。

该资料能够把最简的 `ELF` 文件（简单返回一个数值）压缩到 45 个字节，真地是非常极端的努力，思路可以充分借鉴。在充分理解原文的基础上，我们进行更细致地梳理。

首先对 `ELF` 文件头部和程序头部做更彻底的理解，并具体到每一个字节的含义以及在 Linux 系统下的实际解析情况。

先来看看 `readelf -a` 的结果：

```
$ as --32 -o hello.o hello.s
$ ld -melf_i386 -o hello hello.o
$ sstrip hello
$ readelf -a hello
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endi
an
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048054
  Start of program headers:          52 (bytes into file)
  Start of section headers:          0 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         1
  Size of section headers:           0 (bytes)
  Number of section headers:         0
  Section header string table index: 0

There are no sections in this file.

There are no sections to group in this file.

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz
Flg Align
  LOAD           0x000000 0x08048000 0x08048000 0x00060 0x00060
R E 0x1000
```

然后结合 `/usr/include/linux/elf.h` 分别做详细注解。

首先是 52 字节的 `Elf` 文件头的结构体 `elf32_hdr` ：

变量类型	变量名	字节	说明	类型
unsigned char	e_ident[EI_NIDENT]	16	.ELF 前四个标识文件类型	必须
Elf32_Half	e_type	2	指定为可执行文件	必须
Elf32_Half	e_machine	2	指示目标机类型，例如：Intel 386	必须
Elf32_Word	e_version	4	当前只有一个版本存在，被忽略了	可篡改
Elf32_Addr	e_entry	4	代码入口=加载地址(p_vaddr+.text偏移)	可调整
Elf32_Off	e_phoff	4	程序头 Phdr 的偏移地址，用于加载代码	必须
Elf32_Off	e_shoff	4	所有节区相关信息对文件执行无效	可篡改
Elf32_Word	e_flags	4	Intel 架构未使用	可篡改
Elf32_Half	e_ehsize	2	文件头大小，Linux 没做校验	可篡改
Elf32_Half	e_phentsize	2	程序头入口大小，新内核有用	必须
Elf32_Half	e_phnum	2	程序头入口个数	必须
Elf32_Half	e_shentsize	2	所有节区相关信息对文件执行无效	可篡改
Elf32_Half	e_shnum	2	所有节区相关信息对文件执行无效	可篡改
Elf32_Half	e_shstrndx	2	所有节区相关信息对文件执行无效	可篡改

其次是 32 字节的程序头（Phdr）的结构体 `elf32_phdr` ：

变量类型	变量名	字节	说明	类型
Elf32_Word	p_type	4	标记为可加载段	必须
Elf32_Off	p_offset	4	相对程序头的偏移地址	必须
Elf32_Addr	p_vaddr	4	加载地址, 0x0~0x80000000，页对齐	可调整
Elf32_Addr	p_paddr	4	物理地址，暂时没用	~~可篡改~~
Elf32_Word	p_filesz	4	加载的文件大小，>=real size	可调整
Elf32_Word	p_memsz	4	加载所需内存大小，>= p_filesz	可调整
Elf32_Word	p_flags	4	权限:read(4),exec(1), 其中一个暗指另外一个	可调整
Elf32_Word	p_align	4	PIC(共享库需要)，对执行文件无效	~~可篡改~~

接着，咱们把 Elf 中的文件头和程序头部分可调整和~~可篡改~~的字节（52 + 32 = 84 个）全部用特别的字体标记出来。

```
$ hexdump -C hello -n 84
```

> 00000000 7f 45 4c 46 ~~01 01 01 00 00 00 00 00 00 00 00 00~~
>
> 00000010 02 00 03 00 ~~01 00 00 00~~ **54 80 04 08 34** 00 00 00
>
> 00000020 ~~84 00 00 00 00 00 00 00~~ 34 00 20 00 01 00 ~~28 00~~
>
> 00000030 ~~05 00 02 00~~|01 00 00 00 00 00 00 00 **00 80 04 08**
>
> 00000040 ~~00 80 04 08~~ **60 00 00 00 60 00 00 00 05 00 00 00**
>
> 00000050 ~~00 10 00 00~~
>
> 00000054

上述 `|` 线之前为文件头，之后为程序头，之前的 `000000xx` 为偏移地址。

如果要把程序头彻底合并进文件头。从上述信息综合来看，文件头有 4 处必须保留，结合资料 [1]，经过对比发现，如果把第 4 行开始的程序头往上平移 3 行，也就是：

> 00000000 ========= ~~01 01 01 00 00 00 00 00 00 00 00 00~~
>
> 00000010 02 00 03 00 ~~01 00 00 00~~ **54 80 04 08 34** 00 00 00
>
> 00000020 ~~84 00 00 00~~
>
> 00000030 ========= 01 00 00 00 00 00 00 00 **00 80 04 08**
>
> 00000040 ~~00 80 04 08~~ **60 00 00 00 60 00 00 00 05 00 00 00**
>
> 00000050 ~~00 10 00 00~~
>
> 00000054

把可直接合并的先合并进去，效果如下：

（文件头）

> 00000000 ========= 01 00 00 00 00 00 00 00 **00 80 04 08** (^^ p_vaddr)
>
> 00000010 02 00 03 00 ~~60 00 00 00~~ **54 80 04 08 34** 00 00 00
>
> 00000020 ================== ^^ e_entry ^^ e_phoff

（程序头）

> 00000030 ========= 01 00 00 00 00 00 00 00 **00 80 04 08** (^^ p_vaddr)
>
> 00000040 02 00 03 00 **60 00 00 00 60 00 00 00 05 00 00 00**
>
> 00000050 ========= ^^ p_filesz ^^ p_memsz ^^p_flags
>
> 00000054

接着需要设法处理好可调整的 6 处，可以逐个解决，从易到难。

- 首先，合并 `e_phoff` 与 `p_flags`

在合并程序头以后，程序头的偏移地址需要修改为 4，即文件的第 4 个字节开始，也就是说 `e_phoff` 需要修改为 04。

而恰好，`p_flags` 的 `read(4)` 和 `exec(1)` 可以只选其一，所以，只保留 `read(4)` 即可，刚好也为 04。

合并后效果如下：

（文件头）

> 00000000 ========= 01 00 00 00 00 00 00 00 **00 80 04 08** (^^ p_vaddr)
>
> 00000010 02 00 03 00 ~~60 00 00 00~~ **54 80 04 08** 04 00 00 00
>
> 00000020 =================== ^^ e_entry

（程序头）

> 00000030 ========= 01 00 00 00 00 00 00 00 **00 80 04 08** (^^ p_vaddr)
>
> 00000040 02 00 03 00 **60 00 00 00 60 00 00 00** 04 00 00 00
>
> 00000050 ========= ^^ p_filesz ^^ p_memsz
>
> 00000054

- 接下来，合并 `e_entry` ，`p_filesz`, `p_memsz` 和 `p_vaddr`

从早前的分析情况来看，这 4 个变量基本都依赖 `p_vaddr` ，也就是程序的加载地址，大体的依赖关系如下：

```
e_entry = p_vaddr + text offset = p_vaddr + 84 = p_vaddr + 0x54

p_memsz = e_entry

p_memsz >= p_filesz，可以简单取 p_filesz = p_memsz

p_vaddr = page alignment
```

所以，首先需要确定 `p_vaddr` ，通过测试，发现 `p_vaddr` 最低必须有 64k，也就是 0x00010000，对应到 `hexdump` 的 `little endian` 导出结果，则为 `00 00 01 00` 。

需要注意的是，为了尽量少了分配内存，我们选择了一个最小的 `p_vaddr` ，如果申请的内存太大，系统将无法分配。

接着，计算出另外 3 个变量：

```
e_entry = 0x00010000 + 0x54 = 0x00010054 即 54 00 01 00
p_memsz = 54 00 01 00
p_filesz = 54 00 01 00
```

完全合并后，修改如下：

（文件头）

> 00000000 ========= 01 00 00 00 00 00 00 00 00 00 01 00
>
> 00000010 02 00 03 00 54 00 01 00 54 00 01 00 04 00 00 00
>
> 00000020 ========

好了，直接把内容烧入：

```
$ echo -ne "\x01\x00\x00\x00\x00\x00\x00\x00" \
       "\x00\x00\x01\x00\x02\x00\x03\x00" \
       "\x54\x00\x01\x00\x54\x00\x01\x00\x04" |\
       tr -d ' ' |\
    dd of=hello bs=1 count=25 seek=4 conv=notrunc
```

截掉代码（52 + 32 + 12 = 96）之后的所有内容，查看效果如下：

```
$ dd if=hello of=hello bs=1 count=1 skip=96 seek=96
$ hexdump -C hello -n 96
00000000  7f 45 4c 46 01 00 00 00  00 00 00 00 00 00 01 00  |.EL
F............|
00000010  02 00 03 00 54 00 01 00  54 00 01 00 04 00 00 00  |...
.T...T.......|
00000020  84 00 00 00 00 00 00 00  34 00 20 00 01 00 28 00  |...
.....4. ...(.|
00000030  05 00 02 00 01 00 00 00  00 00 00 00 00 80 04 08  |...
.............|
00000040  00 80 04 08 60 00 00 00  60 00 00 00 05 00 00 00  |...
.`...`.......|
00000050  00 10 00 00 59 59 b2 05  b0 04 cd 80 b0 01 cd 80  |...
.YY..........|
00000060
```

最后的工作是查看文件头中剩下的~~可篡改~~的内容，并把代码部分合并进去，程序头已经合入，不再显示。

> 00000000 7f 45 4c 46 01 00 00 00 00 00 00 00 00 00 01 00
>
> 00000010 02 00 03 00 54 00 01 00 54 00 01 00 04 00 00 00
>
> 00000020 ~~84 00 00 00 00 00 00 00~~ 34 00 20 00 01 00 ~~28 00~~
>
> 00000030 ~~05 00 02 00~~
>
> 00000040
>
> 00000050 ============= **59 59 b2 05 b0 04 cd 80 b0 01 cd 80**
>
> 00000060

我们的指令有 12 字节，~~可篡改~~的部分有 14 个字节，理论上一定放得下，不过因为把程序头搬进去以后，这 14 个字节并不是连续，刚好可以用上我们之前的跳转指令处理办法来解决。

并且，加入 2 个字节的跳转指令，刚好是 14 个字节，恰好把代码也完全包含进了文件头。

在预留好跳转指令位置的前提下，我们把代码部分先合并进去：

> 00000000 7f 45 4c 46 01 00 00 00 00 00 00 00 00 00 01 00
>
> 00000010 02 00 03 00 54 00 01 00 54 00 01 00 04 00 00 00
>
> 00000020 **59 59 b2 05 b0 04** ~~00 00~~ 34 00 20 00 01 00 **cd 80**
>
> 00000030 **b0 01 cd 80**

接下来设计跳转指令，跳转指令需要从所在位置跳到第一个 **cd 80** 所在的位置，相距 6 个字节，根据 `jmp` 短跳转的编码规范，可以设计为 `0xeb 0x06` ，填完后效果如下：

> 00000000 7f 45 4c 46 01 00 00 00 00 00 00 00 00 00 01 00
>
> 00000010 02 00 03 00 54 00 01 00 54 00 01 00 04 00 00 00
>
> 00000020 **59 59 b2 05 b0 04 eb 06** 34 00 20 00 01 00 **cd 80**
>
> 00000030 **b0 01 cd 80**

用 `dd` 命令写入，分两段写入：

```
$ echo -ne "\x59\x59\xb2\x05\xb0\x04\xeb\x06" | \
    dd of=hello bs=1 count=8 seek=32 conv=notrunc

$ echo -ne "\xcd\x80\xb0\x01\xcd\x80" | \
    dd of=hello bs=1 count=6 seek=46 conv=notrunc
```

代码合入以后，需要修改文件头中的代码的偏移地址，即 `e_entry` ，也就是要把原来的偏移 84 (0x54) 修改为现在的偏移，即 0x20。

```
$ echo -ne "\x20" | dd of=hello bs=1 count=1 seek=24 conv=notrun
c
```

修改完以后恰好把合并进的程序头 `p_memsz` ，也就是分配给文件的内存改小了， `p_filesz` 也得相应改小。

```
$ echo -ne "\x20" | dd of=hello bs=1 count=1 seek=20 conv=notrun
c
```

程序头和代码都已经合入，最后，把 52 字节之后的内容全部删掉：

```
$ dd if=hello of=hello bs=1 count=1 skip=52 seek=52
$ hexdump -C hello
00000000  7f 45 4c 46 01 00 00 00  00 00 00 00 00 00 01 00  |.EL
F............|
00000010  02 00 03 00 20 00 01 00  20 00 01 00 04 00 00 00  |...
.T...T.......|
00000020  59 59 b2 05 b0 04 eb 06  34 00 20 00 01 00 cd 80  |YY.
.....4. .....|
00000030  b0 01 cd 80
$ export PATH=./:$PATH
$ hello
hello
```

代码和~~程序头~~部分合并进文件头的汇总情况：

> 00000000 7f 45 4c 46 ~~01 00 00 00 00 00 00 00 00 00 01 00~~
>
> 00000010 ~~02 00 03 00 20 00 01 00 20 00 01 00 04 00 00 00~~
>
> 00000020 ~~**59 59 b2 05**~~ **b0 04 eb 06** 34 00 20 00 01 00 **cd 80**
>
> 00000030 **b0 01 cd 80**

最后，我们的成绩是：

```
$ wc -c hello
52
```

史上最小的可打印 `Hello World` （注：要完全打印得把代码中的5该为13，并且把文件名该为该字符串） 的 `Elf` 文件是 52 个字节。打破了资料 [1] 作者创造的纪录：

```
$ cd ELFkickers/tiny/
$ wc -c hello
59 hello
```

需要特别提到的是，该作者创造的最小可执行 Elf 是 45 个字节。

但是由于那个程序只能返回一个数值，代码更简短，刚好可以直接嵌入到文件头中间，而文件末尾的 7 个 `0` 字节由于 Linux 加载时会自动填充，所以可以删掉，所以最终的文件大小是 52 - 7 即 45 个字节。

其大体可实现如下：

```
.global _start
_start:
    mov $42, %bl   # 设置返回值为 42
    xor %eax, %eax # eax = 0
    inc %eax       # eax = eax+1, 设置系统调用号, sys_exit()
    int $0x80
```

保存为 ret.s，编译和执行效果如下：

```
$ as --32 -o ret.o ret.s
$ ld -melf_i386 -o ret ret.o
$ ./ret
42
```

代码字节数可这么查看：

```
$ ld -melf_i386 --oformat=binary -o ret.bin ret.o
$ hexdump -C ret.bin
0000000  b3 2a 31 c0 40 cd 80
0000007
```

这里只有 7 条指令，刚好可以嵌入，而最后的 6 个字节因为~~可篡改~~为 0，并且内核可自动填充 0，所以干脆可以连续删掉最后 7 个字节的 0：

> 00000000 7f 45 4c 46 01 00 00 00 00 00 00 00 00 00 01 00
>
> 00000010 02 00 03 00 54 00 01 00 54 00 01 00 04 00 00 00
>
> 00000020 **b3 2a 31 c0 40 cd 80** 00 34 00 20 00 01 00 00 00
>
> 00000030 00 00 00 00

可以直接用已经合并好程序头的 `hello` 来做实验，这里一并截掉最后的 7 个 0 字节：

```
$ cp hello ret
$ echo -ne "\xb3\x2a\x31\xc0\x40\xcd\x80" |\
    dd of=ret bs=1 count=8 seek=32 conv=notrunc
$ dd if=ret of=hello bs=1 count=1 skip=45 seek=45
$ hexdump -C hello
00000000  7f 45 4c 46 01 00 00 00  00 00 00 00 00 00 01 00  |.EL
F............|
00000010  02 00 03 00 20 00 01 00  20 00 01 00 04 00 00 00  |...
. ... .......|
00000020  b3 2a 31 c0 40 cd 80 06  34 00 20 00 01           |.*1
.@...4. ..|
0000002d
$ wc -c ret
45 ret
$ ./ret
$ echo $?
42
```

如果想快速构建该 `Elf` 文件，可以直接使用下述 Shell 代码：

```
#!/bin/bash
#
# generate_ret_elf.sh -- Generate a 45 bytes Elf file
#
# $ bash generate_ret_elf.sh
# $ chmod a+x ret.elf
# $ ./ret.elf
# $ echo $?
# 42
#

ret="\x7f\x45\x4c\x46\x01\x00\x00\x00"
ret=${ret}"\x00\x00\x00\x00\x00\x00\x01\x00"
ret=${ret}"\x02\x00\x03\x00\x20\x00\x01\x00"
ret=${ret}"\x20\x00\x01\x00\x04\x00\x00\x00"
ret=${ret}"\xb3\x2a\x31\xc0\x40\xcd\x80\x06"
ret=${ret}"\x34\x00\x20\x00\x01"

echo -ne $ret > ret.elf
```

又或者是直接参照资料 [1] 的 `tiny.asm` 就行了，其代码如下：

```
; ret.asm

  BITS 32

            org     0x00010000

            db      0x7F, "ELF"             ; e_ident
            dd      1                                       ; p_type
            dd      0                                       ; p_offset
            dd      $$                                      ; p_vaddr
            dw      2                       ; e_type        ; p_paddr
            dw      3                       ; e_machine
            dd      _start                  ; e_version     ; p_filesz
            dd      _start                  ; e_entry       ; p_memsz
            dd      4                       ; e_phoff       ; p_flags
  _start:
            mov     bl, 42                  ; e_shoff       ; p_align
            xor     eax, eax
            inc     eax                     ; e_flags
            int     0x80
            db      0
            dw      0x34                    ; e_ehsize
            dw      0x20                    ; e_phentsize
            db      1                       ; e_phnum
                                            ; e_shentsize
                                            ; e_shnum
                                            ; e_shstrndx

  filesize      equ     $ - $$
```

编译和运行效果如下：

```
$ nasm -f bin -o ret ret.asm
$ chmod +x ret
$ ./ret ; echo $?
42
$ wc -c ret
45 ret
```

下面也给一下本文精简后的 `hello` 的 `nasm` 版本：

```
; hello.asm

  BITS 32

            org     0x00010000

            db      0x7F, "ELF"             ; e_ident
            dd      1                                       ; p_
type
            dd      0                                       ; p_
offset
            dd      $$                                      ; p_
vaddr
            dw      2                       ; e_type        ; p_
paddr
            dw      3                       ; e_machine
            dd      _start                  ; e_version     ; p_
filesz
            dd      _start                  ; e_entry       ; p_
memsz
            dd      4                       ; e_phoff       ; p_
flags
  _start:
            pop     ecx     ; argc          ; e_shoff       ; p_
align
            pop     ecx     ; argv[0]
            mov     dl, 5   ; str len       ; e_flags
            mov     al, 4   ; sys_write(fd, addr, len) : ebx, ec
x, edx
            jmp     _next   ; jump to next part of the code
            dw      0x34                      ; e_ehsize
            dw      0x20                      ; e_phentsize
            dw      1                         ; e_phnum
  _next:        int     0x80    ; syscall         ; e_shentsize
            mov     al, 1   ; eax=1,sys_exit  ; e_shnum
            int     0x80    ; syscall         ; e_shstrndx

  filesize      equ     $ - $$
```

编译和用法如下：

```
$ nasm -f bin -o hello hello.asm
$ chmod a+x hello
$ export PATH=./:$PATH
$ hello
hello
$ wc -c hello
52
```

经过一番努力，`AT&T` 的完整 binary 版本如下：

```
# hello.s
#
# as --32 -o hello.o hello.s
# ld -melf_i386 --oformat=binary -o hello hello.o
#

    .file "hello.s"
    .global _start, _load
    .equ   LOAD_ADDR, 0x00010000   # Page aligned load addr, her
e 64k
    .equ   E_ENTRY, LOAD_ADDR + (_start - _load)
    .equ   P_MEM_SZ, E_ENTRY
    .equ   P_FILE_SZ, P_MEM_SZ

_load:
    .byte  0x7F
    .ascii "ELF"                  # e_ident, Magic Number
    .long  1                                      # p_type, load
able seg
    .long  0                                      # p_offset
    .long  LOAD_ADDR                              # p_vaddr
    .word  2                      # e_type, exec  # p_paddr
    .word  3                      # e_machine, Intel 386 target
    .long  P_FILE_SZ              # e_version     # p_filesz
    .long  E_ENTRY                # e_entry       # p_memsz
    .long  4                      # e_phoff       # p_flags, rea
d(exec)
```

```
    .text
_start:
    popl   %ecx    # argc          # e_shoff       # p_align
    popl   %ecx    # argv[0]
    mov    $5, %dl # str len       # e_flags
    mov    $4, %al # sys_write(fd, addr, len) : ebx, ecx, edx
    jmp    next    # jump to next part of the code
    .word  0x34                    # e_ehsize = 52
    .word  0x20                    # e_phentsize = 32
    .word  1                       # e_phnum = 1
    .text
_next: int    $0x80   # syscall         # e_shentsize
    mov    $1, %al # eax=1,sys_exit # e_shnum
    int    $0x80   # syscall        # e_shstrndx
```

编译和运行效果如下：

```
$ as --32 -o hello.o hello.s
$ ld -melf_i386 --oformat=binary -o hello hello.o
$ export PATH=./:$PATH
$ hello
hello
$ wc -c hello
52 hello
```

注：编译时务必要加 `--oformat=binary` 参数，以便直接基于源文件构建一个二进制的 `Elf` 文件，否则会被 `ld` 默认编译，自动填充其他内容。

汇编语言极限精简之道（45字节）

经过上述努力，我们已经完全把程序头和代码都融入了 52 字节的 `Elf` 文件头，还可以再进一步吗？

基于资料一，如果再要努力，只能设法把 `Elf` 末尾的 7 个 0 字节删除，但是由于代码已经把 `Elf` 末尾的 7 字节 0 字符都填满了，所以要想在这一块努力，只能继续压缩代码。

继续研究下代码先：

```
.global _start
_start:
    popl %ecx    # argc
    popl %ecx    # argv[0]
    movb $5, %dl    # 设置字符串长度
    movb $4, %al    # eax = 4, 设置系统调用号, sys_write(fd, addr,
 len) : ebx, ecx, edx
    int $0x80
    movb $1, %al
    int $0x80
```

查看对应的编码：

```
$ as --32 -o hello.o hello.s
$ ld -melf_i386 -o hello hello.o --oformat=binary
$ hexdump -C hello
00000000  59 59 b2 05 b0 04 cd 80  b0 01 cd 80              |YY.
.........|
0000000c
```

每条指令对应的编码映射如下：

指令	编码	说明
popl %ecx	59	argc
popl %ecx	59	argv[0]
movb $5, %dl	b2 05	设置字符串长度
movb $4, %al	b0 04	eax = 4, 设置系统调用号, sys_write(fd, addr, len) : ebx, ecx, edx
int $0x80	cd 80	触发系统调用
movb $1, %al	b0 01	eax = 1, sys_exit
int $0x80	cd 80	触发系统调用

可以观察到：

- `popl` 的指令编码最简洁。
- `int $0x80` 重复了两次，而且每条都占用了 2 字节
- `movb` 每条都占用了 2 字节
- `eax` 有两次赋值，每次占用了 2 字节
- `popl %ecx` 取出的 argc 并未使用

根据之前通过参数传递字符串的想法，咱们是否可以考虑通过参数来设置变量呢？

理论上，传入多个参数，通过 `pop` 弹出来赋予 `eax`, `ecx` 即可，但是实际上，由于从参数栈里头 `pop` 出来的参数是参数的地址，并不是参数本身，所以该方法行不通。

不过由于第一个参数取出的是数字，并且是参数个数，而且目前的那条 `popl %ecx` 取出的 `argc` 并没有使用，那么刚好可以用来设置 `eax`，替换后如下：

```
.global _start
_start:
    popl %eax    # eax = 4, 设置系统调用号, sys_write(fd, addr, le
n) : ebx, ecx, edx
    popl %ecx    # argv[0], 字符串
    movb $5, %dl # 设置字符串长度
    int $0x80
    movb $1, %al # eax = 1, sys_exit
    int $0x80
```

这里需要传入 4 个参数，即让栈弹出的第一个值，也就是参数个数赋予 `eax`，也就是：`hello 5 4 1`。

难道我们只能把该代码优化到 10 个字节？

巧合地是，当偶然改成这样的情况下，该代码还能正常返回。

```
.global _start
_start:
    popl %eax    # eax = 4, 设置系统调用号, sys_write(fd, addr, le
n) : ebx, ecx, edx
    popl %ecx    # argv[0], 字符串
    movb $5, %dl    # 设置字符串长度
    int $0x80
    loop _start     # 触发系统退出
```

注：上面我们使用了 `loop` 指令而不是 `jmp` 指令，因为 `jmp _start` 产生的代码更长，而 `loop _start` 指令只有两个字节。

这里相当于删除了 `movb $1, %al` ，最后我们获得了 8 个字节。但是这里为什么能够工作呢？

经过分析 `arch/x86/ia32/ia32entry.S` ，我们发现当系统调用号无效时（超过系统调用入口个数），内核为了健壮考虑，必须要处理这类异常，并通过 `ia32_badsys` 让系统调用正常返回。

这个可以这样验证：

```
.global _start
_start:
    popl %eax    # argc, eax = 4, 设置系统调用号, sys_write(fd, ad
dr, len) : ebx, ecx, edx
    popl %ecx    # argv[0], 文件名
    mov $5, %dl  # argv[1]，字符串长度
    int $0x80
    mov $0xffffffda, %eax  # 设置一个非法调用号用于退出
    int $0x80
```

那最后的结果是，我们产生了一个可以正常打印字符串，大小只有 45 字节的 `Elf` 文件，最终的结果如下：

```
# hello.s
#
# $ as --32 -o hello.o hello.s
# $ ld -melf_i386 --oformat=binary -o hello hello.o
```

```
# $ export PATH=./:$PATH
# $ hello 0 0 0
# hello
#

    .file "hello.s"
    .global _start, _load
    .equ   LOAD_ADDR, 0x00010000   # Page aligned load addr, her
e 64k
    .equ   E_ENTRY, LOAD_ADDR + (_start - _load)
    .equ   P_MEM_SZ, E_ENTRY
    .equ   P_FILE_SZ, P_MEM_SZ

_load:
    .byte  0x7F
    .ascii "ELF"              # e_ident, Magic Number
    .long  1                                      # p_type, load
able seg
    .long  0                                      # p_offset
    .long  LOAD_ADDR                              # p_vaddr
    .word  2                  # e_type, exec  # p_paddr
    .word  3                  # e_machine, Intel 386 target
    .long  P_FILE_SZ          # e_version     # p_filesz
    .long  E_ENTRY            # e_entry       # p_memsz
    .long  4                  # e_phoff       # p_flags, read(ex
ec)
    .text
_start:
    popl   %eax    # argc     # e_shoff       # p_align
                   # 4 args, eax = 4, sys_write(fd, addr, len) :
 ebx, ecx, edx
                   # set 2nd eax = random addr to trigger bad sy
scall for exit
    popl   %ecx    # argv[0]
    mov    $5, %dl # str len  # e_flags
    int    $0x80
    loop   _start  # loop to popup a random addr as a bad syscal
l number
    .word  0x34               # e_ehsize = 52
    .word  0x20               # e_phentsize = 32
```

```
    .byte  1                    # e_phnum = 1, remove trailing 7 b
ytes with 0 value
                                # e_shentsize
                                # e_shnum
                                # e_shstrndx
```

效果如下：

```
$ as --32 -o hello.o hello.s
$ ld -melf_i386 -o hello hello.o --oformat=binary
$ export PATH=./:$PATH
$ hello 0 0 0
hello
$ wc -c hello
45 hello
```

到这里，我们获得了史上最小的可以打印字符串的 `Elf` 文件，是的，只有 45 个字节。

小结

到这里，关于可执行文件的讨论暂且结束，最后来一段小小的总结，那就是我们设法去减少可执行文件大小的意义？

实际上，通过这样一个讨论深入到了很多技术的细节，包括可执行文件的格式、目标代码链接的过程、 Linux 下汇编语言开发等。与此同时，可执行文件大小的减少本身对嵌入式系统非常有用，如果删除那些对程序运行没有影响的节区和节区表将减少目标系统的大小，适应嵌入式系统资源受限的需求。除此之外，动态连接库中的很多函数可能不会被使用到，因此也可以通过某种方式剔除 [8]，[10] 。

或许，你还会发现更多有趣的意义，欢迎给我发送邮件，一起讨论。

参考资料

- A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux
- UNIX/LINUX 平台可执行文件格式分析
- C/C++ 程序编译步骤详解

- The Linux GCC HOW TO
- ELF: From The Programmer's Perspective
- Understanding ELF using readelf and objdump
- Dissecting shared libraries
- 嵌入式 Linux 小型化技术
- Linux 汇编语言开发指南
- Library Optimizer
- ELF file format and ABI：[1]，[2]，[3]，[4]
- i386 指令编码表

代码测试、调试与优化

前言

代码写完以后往往要做测试（或验证）、调试，可能还要优化。

- 关于测试（或验证）

    通常对应着两个英文单词 `Verification` 和 `Validation`，在资料 [1] 中有关于这个的定义和一些深入的讨论，在资料 [2] 中，很多人给出了自己的看法。但是正如资料 [2] 提到的：

> The differences between verification and validation are unimportant except to the theorist; practitioners use the term V&V to refer to all of the activities that are aimed at making sure the software will function as required.

所以，无论测试（或验证）目的都是为了让软件的功能能够达到需求。测试和验证通常会通过一些形式化（貌似可以简单地认为有数学根据的）或者非形式化的方法去验证程序的功能是否达到要求。

- 关于调试

  而调试对应英文 debug，debug 叫“驱除害虫”，也许一个软件的功能达到了要求，但是可能会在测试或者是正常运行时出现异常，因此需要处理它们。

- 关于优化

  debug 是为了保证程序的正确性，之后就需要考虑程序的执行效率，对于存储资源受限的嵌入式系统，程序的大小也可能是优化的对象。

  很多理论性的东西实在没有研究过，暂且不说吧。这里只是想把一些需要动手实践的东西先且记录和总结一下，另外很多工具在这里都有提到和罗列，包括 Linux 内核调试相关的方法和工具。关于更详细更深入的内容还是建议直接看后面的参考资料为妙。

下面的所有演示在如下环境下进行：

```
$ uname -a
Linux falcon 2.6.22-14-generic #1 SMP Tue Feb 12 07:42:25 UTC 20
08 i686 GNU/Linux
$ echo $SHELL
/bin/bash
$ /bin/bash --version | grep bash
GNU bash, version 3.2.25(1)-release (i486-pc-linux-gnu)
$ gcc --version | grep gcc
gcc (GCC) 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)
$ cat /proc/cpuinfo | grep "model name"
model name      : Intel(R) Pentium(R) 4 CPU 2.80GHz
```

代码测试

代码测试有很多方面，例如运行时间、函数调用关系图、代码覆盖度、性能分析（Profiling）、内存访问越界（Segmentation Fault）、缓冲区溢出（Stack Smashing 合法地进行非法的内存访问？所以很危险）、内存泄露（Memory Leak）等。

测试程序的运行时间 time

Shell 提供了内置命令 `time` 用于测试程序的执行时间，默认显示结果包括三部分：实际花费时间（real time）、用户空间花费时间（user time）和内核空间花费时间（kernel time）。

```
$ time pstree 2>&1 >/dev/null

real    0m0.024s
user    0m0.008s
sys     0m0.004s
```

`time` 命令给出了程序本身的运行时间。这个测试原理非常简单，就是在程序运行（通过 `system` 函数执行）前后记录了系统时间（用 `times` 函数），然后进行求差就可以。如果程序运行时间很短，运行一次看不到效果，可以考虑采用测试纸片厚度的方法进行测试，类似把很多纸张叠到一起来测试纸张厚度一样，我们可以让程序运行很多次。

如果程序运行时间太长，执行效率很低，那么得考虑程序内部各个部分的执行情况，从而对代码进行可能的优化。具体可能会考虑到这两点：

对于 C 语言程序而言，一个比较宏观的层次性的轮廓（profile）是函数调用图、函数内部的条件分支构成的语句块，然后就是具体的语句。把握好这样一个轮廓后，就可以有针对性地去关注程序的各个部分，包括哪些函数、哪些分支、哪些语句最值得关注（执行次数越多越值得优化，术语叫 hotspots）。

对于 Linux 下的程序而言，程序运行时涉及到的代码会涵盖两个空间，即用户空间和内核空间。由于这两个空间涉及到地址空间的隔离，在测试或调试时，可能涉及到两个空间的工具。前者绝大多数是基于 `Gcc` 的特定参数和系统的 `ptrace` 调用，而后者往往实现为内核的补丁，它们在原理上可能类似，但实际操作时后者显然会更麻烦，不过如果你不去 hack 内核，那么往往无须关心后者。

函数调用关系图 calltree

`calltree` 可以非常简单方便地反应一个项目的函数调用关系图，虽然诸如 `gprof` 这样的工具也能做到，不过如果仅仅要得到函数调用图， `calltree` 应该是更好的选择。如果要产生图形化的输出可以使用它的 `-dot` 参数。从这里可以下载到它。

这里是一份基本用法演示结果：

```
$ calltree -b -np -m *.c
main:
|   close
|   commitchanges
|   |   err
|   |   |   fprintf
|   |   ferr
|   |   ftruncate
|   |   lseek
|   |   write
|   ferr
|   getmemorysize
|   modifyheaders
|   open
|   printf
|   readelfheader
|   |   err
|   |   |   fprintf
|   |   ferr
|   |   read
|   readphdrtable
|   |   err
|   |   |   fprintf
|   |   ferr
|   |   malloc
|   |   read
|   truncatezeros
|   |   err
|   |   |   fprintf
|   |   ferr
|   |   lseek
|   |   read$
```

这样一份结果对于“反向工程”应该会很有帮助，它能够呈现一个程序的大体结构，对于阅读和分析源代码来说是一个非常好的选择。虽然 `cscope` 和 `ctags` 也能够提供一个函数调用的“即时”（在编辑 Vim 的过程中进行调用）视图（view），但是 `calltree` 却给了我们一个宏观的视图。

不过这样一个视图只涉及到用户空间的函数，如果想进一步给出内核空间的宏观视图，那么 `strace` ， `KFT` 或者 `Ftrace` 就可以发挥它们的作用。另外，该视图也没有给出库中的函数，如果要跟踪呢？需要 `ltrace` 工具。

另外发现 `calltree` 仅仅给出了一个程序的函数调用视图，而没有告诉我们各个函数的执行次数等情况。如果要关注这些呢？我们有 `gprof` 。

性能测试工具 gprof & kprof

参考资料[3]详细介绍了这个工具的用法，这里仅挑选其中一个例子来演示。 `gprof` 是一个命令行的工具，而 KDE 桌面环境下的 `kprof` 则给出了图形化的输出，这里仅演示前者。

首先来看一段代码（来自资料[3]），算 `Fibonacci` 数列的，

```c
#include <stdio.h>

int fibonacci(int n);

int main (int argc, char **argv)
{
    int fib;
    int n;

    for (n = 0; n <= 42; n++) {
        fib = fibonacci(n);
        printf("fibonnaci(%d) = %d\n", n, fib);
    }

    return 0;
}

int fibonacci(int n)
{
    int fib;

    if (n <= 0) {
        fib = 0;
    } else if (n == 1) {
        fib = 1;
    } else {
        fib = fibonacci(n -1) + fibonacci(n - 2);
    }

    return fib;
}
```

通过 `calltree` 看看这段代码的视图，

```
$ calltree -b -np -m *.c
main:
|   fibonacci
|   |   fibonacci ....
|   printf
```

可以看出程序主要涉及到一个 `fibonacci` 函数，这个函数递归调用自己。为了能够使用 `gprof`，需要编译时加上 `-pg` 选项，让 `Gcc` 加入相应的调试信息以便 `gprof` 能够产生函数执行情况的报告。

```
$ gcc -pg -o fib fib.c
$ ls
fib  fib.c
```

运行程序并查看执行时间，

```
$ time ./fib
fibonnaci(0) = 0
fibonnaci(1) = 1
fibonnaci(2) = 1
fibonnaci(3) = 2
...
fibonnaci(41) = 165580141
fibonnaci(42) = 267914296

real    1m25.746s
user    1m9.952s
sys     0m0.072s
$ ls
fib  fib.c  gmon.out
```

上面仅仅选取了部分执行结果，程序运行了 1 分多钟，代码运行以后产生了一个 `gmon.out` 文件，这个文件可以用于 `gprof` 产生一个相关的性能报告。

```
$ gprof  -b ./fib gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 96.04     14.31    14.31       43   332.80   332.80  fibonacci
  4.59     14.99     0.68                             main


                        Call graph


granularity: each sample hit covers 2 byte(s) for 0.07% of 14.99
 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.68   14.31                 main [1]
               14.31    0.00      43/43          fibonacci [2]
-----------------------------------------------
                             2269806252             fibonacci [2
]
               14.31    0.00      43/43          main [1]
[2]     95.4   14.31    0.00      43+2269806252 fibonacci [2]
                             2269806252             fibonacci [2
]
-----------------------------------------------


Index by function name

   [2] fibonacci               [1] main
```

从这份结果中可观察到程序中每个函数的执行次数等情况，从而找出值得修改的函数。在对某些部分修改之后，可以再次比较程序运行时间，查看优化结果。另外，这份结果还包含一个特别有用的东西，那就是程序的动态函数调用情况，即程序运

行过程中实际执行过的函数，这和 `calltree` 产生的静态调用树有所不同，它能够反应程序在该次执行过程中的函数调用情况。而如果想反应程序运行的某一时刻调用过的函数，可以考虑采用 `gdb` 的 `backtrace` 命令。

类似测试纸片厚度的方法，`gprof` 也提供了一个统计选项，用于对程序的多次运行结果进行统计。另外，`gprof` 有一个 KDE 下图形化接口 `kprof`，这两部分请参考资料[3]。

对于非 KDE 环境，可以使用 Gprof2Dot 把 `gprof` 输出转换成图形化结果。

关于 `dot` 格式的输出，也可以可以考虑通过 `dot` 命令把结果转成 `jpg` 等格式，例如：

```
$ dot -Tjpg test.dot -o test.jp
```

`gprof` 虽然给出了函数级别的执行情况，但是如果想关心具体哪些条件分支被执行到，哪些语句没有被执行，该怎么办？

代码覆盖率测试 gcov & ggcov

如果要使用 `gcov`，在编译时需要加上这两个选项 `-fprofile-arcs -ftest-coverage`，这里直接用之前的 `fib.c` 做演示。

```
$ ls
fib.c
$ gcc -fprofile-arcs -ftest-coverage -o fib fib.c
$ ls
fib  fib.c  fib.gcno
```

运行程序，并通过 `gcov` 分析代码的覆盖度：

```
$ ./fib
$ gcov fib.c
File 'fib.c'
Lines executed:100.00% of 12
fib.c:creating 'fib.c.gcov'
```

12 行代码 100% 被执行到，再查看分支情况，

```
$ gcov -b fib.c
File 'fib.c'
Lines executed:100.00% of 12
Branches executed:100.00% of 6
Taken at least once:100.00% of 6
Calls executed:100.00% of 4
fib.c:creating 'fib.c.gcov'
```

发现所有函数，条件分支和语句都被执行到，说明代码的覆盖率很高，不过资料[3] `gprof` 的演示显示代码的覆盖率高并不一定说明代码的性能就好，因为那些被覆盖到的代码可能能够被优化成性能更高的代码。那到底哪些代码值得被优化呢？执行次数最多的，另外，有些分支虽然都覆盖到了，但是这个分支的位置可能并不是理想的，如果一个分支的内容被执行的次数很多，那么把它作为最后一个分支的话就会浪费很多不必要的比较时间。因此，通过覆盖率测试，可以尝试着剔除那些从未执行过的代码或者把那些执行次数较多的分支移动到较早的条件分支里头。通过性能测试，可以找出那些值得优化的函数、分支或者是语句。

如果使用 `-fprofile-arcs -ftest-coverage` 参数编译完代码，可以接着用 `-fbranch-probabilities` 参数对代码进行编译，这样，编译器就可以对根据代码的分支测试情况进行优化。

```
$ wc -c fib
16333 fib
$ ls fib.gcda  #确保fib.gcda已经生成，这个是运行fib后的结果
fib.gcda
$ gcc -fbranch-probabilities -o fib fib.c #再次运行
$ wc -c fib
6604 fib
$ time ./fib
...
real    0m21.686s
user    0m18.477s
sys     0m0.008s
```

可见代码量减少了，而且执行效率会有所提高，当然，这个代码效率的提高可能还跟其他因素有关，比如 `Gcc` 还优化了一些跟平台相关的指令。

如果想看看代码中各行被执行的情况，可以直接看 `fib.c.gcov` 文件。这个文件的各列依次表示执行次数、行号和该行的源代码。次数有三种情况，如果一直没有执行，那么用 `#####` 表示；如果该行是注释、函数声明等，用 `-` 表示；如果是纯粹的代码行，那么用执行次数表示。这样我们就可以直接分析每一行的执行情况。

`gcov` 也有一个图形化接口 `ggcov`，是基于 `gtk+` 的，适合 Gnome 桌面的用户。

现在都已经关注到代码行了，实际上优化代码的前提是保证代码的正确性，如果代码还有很多 bug，那么先要 debug。不过下面的这些 "bug" 用普通的工具确实不太方便，虽然可能，不过这里还是把它们归结为测试的内容，并且这里刚好承接上 `gcov` 部分，`gcov` 能够测试到每一行的代码覆盖情况，而无论是内存访问越界、缓冲区溢出还是内存泄露，实际上是发生在具体的代码行上的。

内存访问越界 catchsegv, libSegFault.so

"Segmentation fault" 是很头痛的一个问题，估计“纠缠”过很多人。这里仅仅演示通过 `catchsegv` 脚本测试段错误的方法，其他方法见后面相关资料。

`catchsegv` 利用系统动态链接的 `PRELOAD` 机制（请参考`man ld-linux`），把库 `/lib/libSegFault.so` 提前 load 到内存中，然后通过它检查程序运行过程中的段错误。

```
$ cat test.c
#include <stdio.h>

int main(void)
{
    char str[10];

        sprintf(str, "%s", 111);

        printf("str = %s\n", str);
        return 0;
}
$ make test
$ LD_PRELOAD=/lib/libSegFault.so ./test  #等同于catchsegv ./test
*** Segmentation fault
Register dump:

 EAX: 0000006f   EBX: b7eecff4   ECX: 00000003   EDX: 0000006f
 ESI: 0000006f   EDI: 0804851c   EBP: bff9a8a4   ESP: bff9a27c

 EIP: b7e1755b   EFLAGS: 00010206

 CS: 0073   DS: 007b   ES: 007b   FS: 0000   GS: 0033   SS: 007b

 Trap: 0000000e   Error: 00000004   OldMask: 00000000
 ESP/signal: bff9a27c   CR2: 0000006f

Backtrace:
/lib/libSegFault.so[0xb7f0604f]
[0xffffe420]
/lib/tls/i686/cmov/libc.so.6(vsprintf+0x8c)[0xb7e0233c]
/lib/tls/i686/cmov/libc.so.6(sprintf+0x2e)[0xb7ded9be]
./test[0x804842b]
/lib/tls/i686/cmov/libc.so.6(__libc_start_main+0xe0)[0xb7dbd050]
./test[0x8048391]
...
```

从结果中可以看出，代码的 `sprintf` 有问题。经过检查发现它把整数当字符串输出，对于字符串的输出，需要字符串的地址作为参数，而这里的 `111` 则刚好被解释成了字符串的地址，因此 `sprintf` 试图访问 `111` 这个地址，从而发生了非法访问内存的情况，出现 “Segmentation Fault”。

缓冲区溢出 libsafe.so

缓冲区溢出是堆栈溢出（Stack Smashing），通常发生在对函数内的局部变量进行赋值操作时，超出了该变量的字节长度而引起对栈内原有数据（比如 eip，ebp 等）的覆盖，从而引发内存访问越界，甚至执行非法代码，导致系统崩溃。关于缓冲区的详细原理和实例分析见《缓冲区溢出与注入分析》。这里仅仅演示该资料中提到的一种用于检查缓冲区溢出的方法，它同样采用动态链接的 `PRELOAD` 机制提前装载一个名叫 `libsafe.so` 的库，可以从这里获取它，下载后，再解压，编译，得到 `libsafe.so` ，

下面，演示一个非常简单的，但可能存在缓冲区溢出的代码，并演示 `libsafe.so` 的用法。

```
$ cat test.c
$ make test
$ LD_PRELOAD=/path/to/libsafe.so ./test ABCDEFGHIJKLMN
ABCDEFGHIJKLMN
*** stack smashing detected ***: ./test terminated
Aborted (core dumped)
```

资料[7]分析到，如果不能够对缓冲区溢出进行有效的处理，可能会存在很多潜在的危险。虽然 `libsafe.so` 采用函数替换的方法能够进行对这类 Stack Smashing 进行一定的保护，但是无法根本解决问题，alert7 大虾在资料[10]中提出了突破它的办法，资料1111]提出了另外一种保护机制。

内存泄露 Memwatch, Valgrind, mtrace

堆栈通常会被弄在一起叫，不过这两个名词却是指进程的内存映像中的两个不同的部分，栈（Stack）用于函数的参数传递、局部变量的存储等，是系统自动分配和回收的；而堆（heap）则是用户通过 `malloc` 等方式申请而且需要用户自己通过 `free` 释放的，如果申请的内存没有释放，那么将导致内存泄露，进而可能导致

堆的空间被用尽；而如果已经释放的内存再次被释放（double-free）则也会出现非法操作。如果要真正理解堆和栈的区别，需要理解进程的内存映像，请参考《缓冲区溢出与注入分析》

这里演示通过 `Memwatch` 来检测程序中可能存在内存泄露，可以从这里下载到这个工具。使用这个工具的方式很简单，只要把它链接（ld）到可执行文件中去，并在编译时加上两个宏开关`-DMEMWATCH -DMW_STDIO`。这里演示一个简单的例子。

```
$ cat test.c
#include <stdlib.h>
#include <stdio.h>
#include "memwatch.h"

int main(void)
{
    char *ptr1;
    char *ptr2;

    ptr1 = malloc(512);
    ptr2 = malloc(512);

    ptr2 = ptr1;
    free(ptr2);
    free(ptr1);
}
$ gcc -DMEMWATCH -DMW_STDIO test.c memwatch.c -o test
$ cat memwatch.log
============= MEMWATCH 2.71 Copyright (C) 1992-1999 Johan Lindh
=============

Started at Sat Mar  1 07:34:33 2008

Modes: __STDC__ 32-bit mwDWORD==(unsigned long)
mwROUNDALLOC==4 sizeof(mwData)==32 mwDataSize==32

double-free: <4> test.c(15), 0x80517e4 was freed from test.c(14)

Stopped at Sat Mar  1 07:34:33 2008

unfreed: <2> test.c(11), 512 bytes at 0x8051a14         {FE FE F
E FE FE FE FE FE FE FE FE FE FE FE FE FE ................}

Memory usage statistics (global):
 N)umber of allocations made: 2
 L)argest memory usage      : 1024
 T)otal of all alloc() calls: 1024
 U)nfreed bytes totals      : 512
```

通过测试，可以看到有一个 512 字节的空间没有被释放，而另外 512 字节空间却被连续释放两次（double-free）。 `Valgrind` 和 `mtrace` 也可以做类似的工作，请参考资料[4]，[5]和 `mtrace` 的手册。

代码调试

调试的方法很多，调试往往要跟踪代码的运行状态， `printf` 是最基本的办法，然后呢？静态调试方法有哪些，非交互的呢？非实时的有哪些？实时的呢？用于调试内核的方法有哪些？有哪些可以用来调试汇编代码呢？

静态调试：printf + gcc -D（打印程序中的变量）

利用 `Gcc` 的宏定义开关（ `-D` ）和 `printf` 函数可以跟踪程序中某个位置的状态，这个状态包括当前一些变量和寄存器的值。调试时需要用 `-D` 开关进行编译，在正式发布程序时则可把 `-D` 开关去掉。这样做比单纯用 `printf` 方便很多，它可以避免清理调试代码以及由此带来的代码误删除等问题。

```
$ cat test.c
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    int i = 0;

#ifdef DEBUG
        printf("i = %d\n", i);

        int t;
        __asm__ __volatile__ ("movl %%ebp, %0;":"=r"(t)::"%ebp")
;
        printf("ebp = 0x%x\n", t);
#endif

        _exit(0);
}
$ gcc -DDEBUG -g -o test test.c
$ ./test
i = 0
ebp = 0xbfb56d98
```

上面演示了如何跟踪普通变量和寄存器变量的办法。跟踪寄存器变量采用了内联汇编。

不过，这种方式不够灵活，我们无法“即时”获取程序的执行状态，而 `gdb` 等交互式调试工具不仅解决了这样的问题，而且通过把调试器拆分成调试服务器和调试客户端适应了嵌入式系统的调试，另外，通过预先设置断点以及断点处需要收集的程序状态信息解决了交互式调试不适应实时调试的问题。

交互式的调试（动态调试）：gdb（支持本地和远程）/ald（汇编指令级别的调试）

嵌入式系统调试方法 gdbserver/gdb

估计大家已经非常熟悉 GDB（Gnu DeBugger）了，所以这里并不介绍常规的 `gdb` 用法，而是介绍它的服务器／客户（ `gdbserver/gdb` ）调试方式。这种方式非常适合嵌入式系统的调试，为什么呢？先来看看这个：

```
$ wc -c /usr/bin/gdbserver
56000 /usr/bin/gdbserver
$ which gdb
/usr/bin/gdb
$ wc -c /usr/bin/gdb
2557324 /usr/bin/gdb
$ echo "(2557324-56000)/2557324"  | bc -l
.97810210986171482377
```

`gdb` 比 `gdbserver` 大了将近 97%，如果把整个 `gdb` 搬到存储空间受限的嵌入式系统中是很不合适的，不过仅仅 5K 左右的 `gdbserver` 即使在只有 8M Flash 卡的嵌入式系统中也都足够了。所以在嵌入式开发中，我们通常先在本地主机上交叉编译好 `gdbserver/gdb` 。

如果是初次使用这种方法，可能会遇到麻烦，而麻烦通常发生在交叉编译 `gdb` 和 `gdbserver` 时。在编译 `gdbserver/gdb` 前，需要配置(./configure)两个重要的选项：

- `--host` ，指定 gdb/gdbserver 本身的运行平台，
- `--target` ，指定 gdb/gdbserver 调试的代码所运行的平台，

关于运行平台，通过 `$MACHTYPE` 环境变量就可获得，对于 `gdbserver` ，因为要把它复制到嵌入式目标系统上，并且用它来调试目标平台上的代码，因此需要把 `--host` 和 `--target` 都设置成目标平台；而 `gdb` 因为还是运行在本地主机上，但是需要用它调试目标系统上的代码，所以需要把 `--target` 设置成目标平台。

编译完以后就是调试，调试时需要把程序交叉编译好，并把二进制文件复制一份到目标系统上，并在本地需要保留一份源代码文件。调试过程大体如下，首先在目标系统上启动调试服务器：

```
$ gdbserver :port /path/to/binary_file
...
```

然后在本地主机上启动gdb客户端链接到 `gdb` 调试服务器，（`gdbserver_ipaddress` 是目标系统的IP地址，如果目标系统不支持网络，那么可以采用串口的方式，具体看手册）

```
$ gdb -q
(gdb) target remote gdbserver_ipaddress:2345
...
```

其他调试过程和普通的gdb调试过程类似。

汇编代码的调试 ald

用 `gdb` 调试汇编代码貌似会比较麻烦，不过有人正是因为这个原因而开发了一个专门的汇编代码调试器，名字就叫做 `assembly language debugger`，简称 `ald`，你可以从这里下载到。

下载后，解压编译，我们来调试一个程序看看。

这里是一段非常简短的汇编代码：

```
.global _start
_start:
        popl %ecx
        popl %ecx
        popl %ecx
        movb $10,12(%ecx)
        xorl %edx, %edx
        movb $13, %dl
        xorl %eax, %eax
        movb $4, %al
        xorl %ebx, %ebx
        int $0x80
        xorl %eax, %eax
        incl %eax
        int $0x80
```

汇编、链接、运行：

```
$ as -o test.o test.s
$ ld -o test test.o
$ ./test "Hello World"
Hello World
```

查看程序的入口地址：

```
$ readelf -h test | grep Entry
  Entry point address:               0x8048054
```

接着用 `ald` 调试：

```
$ ald test
ald> display
Address 0x8048054 added to step display list
ald> n
eax = 0x00000000 ebx = 0x00000000 ecx = 0x00000001 edx = 0x00000
000
esp = 0xBFBFDEB4 ebp = 0x00000000 esi = 0x00000000 edi = 0x00000
000
ds  = 0x007B es  = 0x007B fs  = 0x0000 gs  = 0x0000
ss  = 0x007B cs  = 0x0073 eip = 0x08048055 eflags = 0x00200292

Flags: AF SF IF ID

Dumping 64 bytes of memory starting at 0x08048054 in hex
08048054:  59 59 59 C6 41 0C 0A 31 D2 B2 0D 31 C0 B0 04 31    YY
Y.A..1...1...1
08048064:  DB CD 80 31 C0 40 CD 80 00 2E 73 79 6D 74 61 62    ..
.1.@....symtab
08048074:  00 2E 73 74 72 74 61 62 00 2E 73 68 73 74 72 74    ..
strtab..shstrt
08048084:  61 62 00 2E 74 65 78 74 00 00 00 00 00 00 00 00    ab
..text........

08048055                      59                   pop ecx
```

可见 `ald` 在启动时就已经运行了被它调试的 `test` 程序，并且进入了程序的入口 `0x8048054` ，紧接着单步执行时，就执行了程序的第一条指令 `popl ecx` 。

`ald` 的命令很少，而且跟 `gdb` 很类似，比如这个几个命令用法和名字都类似 `help,next,continue,set args,break,file,quit,disassemble,enable,disable` 等。名字不太一样但功能对等的有： `examine` 对 `x` ， `enter` 对 `set variable {int} 地址=数据` 。

需要提到的是：Linux 下的调试器包括上面的 `gdb` 和 `ald` ，以及 `strace` 等都用到了 Linux 系统提供的 ptrace() 系统调用，这个调用为用户访问内存映像提供了便利，如果想自己写一个调试器或者想hack一下 `gdb` 和 `ald` ，那么好好阅读资料12和 `man ptrace` 吧。

如果确实需要用gdb调试汇编，可以参考：

- Linux Assembly "Hello World" Tutorial, CS 200
- Debugging your Alpha Assembly Programs using GDB

实时调试：gdb tracepoint

对于程序状态受时间影响的程序，用上述普通的设置断点的交互式调试方法并不合适，因为这种方式将由于交互时产生的通信延迟和用户输入命令的时延而完全改变程序的行为。所以 `gdb` 提出了一种方法以便预先设置断点以及在断点处需要获取的程序状态，从而让调试器自动执行断点处的动作，获取程序的状态，从而避免在断点处出现人机交互产生时延改变程序的行为。

这种方法叫 `tracepoints` （对应 `breakpoint` ），它在 `gdb` 的用户手册里头有详细的说明，见 Tracepoints。

在内核中，有实现了相应的支持，叫 KGTP。

调试内核

虽然这里并不会演示如何去 hack 内核，但是相关的工具还是需要简单提到的，这个资料列出了绝大部分用于内核调试的工具，这些对你 hack 内核应该会有帮助的。

代码优化

这部分暂时没有准备足够的素材，有待进一步完善。

暂且先提到两个比较重要的工具，一个是 Oprofile，另外一个是 Perf。

实际上呢？“代码测试”部分介绍的很多工具是为代码优化服务的，更多具体的细节请参考后续资料，自己做实验吧。

参考资料

- VERIFICATION AND VALIDATION
- difference between verification and Validation
- Coverage Measurement and Profiling(覆盖度测量和性能测试,Gcov and Gprof)
- Valgrind Usage
    - Valgrind HOWTO
    - Using Valgrind to Find Memory Leaks and Invalid Memory Use
- MEMWATCH
- Mastering Linux debugging techniques
- Software Performance Analysis
- Runtime debugging in embedded systems
- 绕过libsafe的保护--覆盖_dl_lookup_versioned_symbol技术
- 介绍Propolice怎样保护stack-smashing的攻击
- Tools Provided by System：ltrace,mtrace,strace
- Process Tracing Using Ptrace
- Kernel Debugging Related Tools：KGDB, KGOV, KFI/KFT/Ftrace, GDB Tracepoint，UML, kdb
- 用Graphviz 可视化函数调用
- Linux 段错误详解
- 源码分析之函数调用关系绘制系列
    - 源码分析：静态分析 C 程序函数调用关系图
    - 源码分析：动态分析 C 程序函数调用关系
    - 源码分析：动态分析 Linux 内核函数调用关系
    - 源码分析：函数调用关系绘制方法与逆向建模
- Linux 下缓冲区溢出攻击的原理及对策
- Linux 汇编语言开发指南
- 缓冲区溢出与注入分析(第一部分：进程的内存映像)

- Optimizing C Code
- Performance programming for scientific computing
- Performance Programming
- Linux Profiling and Optimization
- High-level code optimization
- Code Optimization